UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Dpto. de TECNOLOGÍA ELECTRÓNICA

INGENIERÍA INDUSTRIAL SUPERIOR

ESPECIALIDAD AUTOMÁTICA Y ELECTRÓNICA INDUSTRIAL

PROYECTO FIN DE CARRERA

ENTORNO DE SIMULACIÓN ACELERADA POR HARDWARE PARA MODULADORES SIGMA-DELTA DE TIEMPO CONTINUO UTILIZANDO UNA FPGA

AUTORA: EVA JUEZ MORALES

TUTORAS: MARTA PORTELA GARCÍA Y SUSANA PATÓN ÁLVAREZ

AGRADECIMIENTOS:

En primer lugar, me gustaría mostrar mi agradecimiento a mis tutoras de proyecto, Marta Portela y Susana Patón, por su paciencia y ayuda a lo largo del proyecto. Aunque también debería darles las gracias por ser unas profesoras tan buenas en las asignaturas que impartieron, ya que sin ellas no habría conseguido llegar hasta donde estoy ahora.

A mis compañeros y amigos de la universidad, que me han ayudado a que estos años fueran más llevaderos y no me sintiera sola en este mundo loco que es la univerdiad.

A mis amigos, especialmente a mi amiga Ana, que siempre ha estado hay en los malos y en los buenos momentos para darme ánimos cuando más lo necesitaba. Y a mi pareja, que ha tenido que soportar años de oir mis quejas sin rechistar.

Por último un agradecimiento especial a mis padres y a mi familia, que me han soportado y apoyado en todas mis decisiones. Y a la comida de mi Tia Pili durante mis primeros años de universidad, que me dio fuerzas en aquellos días en los que no me quedaba tiempo ni para rascarme la nariz.

ÍNDICE:

ÍNDICE DE FIGURAS:

ÍNDICE DE TABLAS:

1 INTRODUCCIÓN

1.1 Introducción.

Este proyecto tiene como principal objetivo la implementación de un modelo hardware de simulación para moduladores Sigma-Delta de Tiempo Continuo que permite acelerar el proceso de simulación de estos circuitos utilizando una FPGA (Field Programmable Gate Array), basándonos en un estudio realizado por Timon Brückner, Matthias Lorenz, Christoph Zorn, Joachim Becker, Wolgang Mathis y Maurits Ortmanns [1].

Un modulador Sigma-Delta de Tiempo Continuo constituye una técnica de conversión de señal analógica a señal digital que resulta ser una alternativa interesante para aplicaciones de alta velocidad, aun a pesar de que la simulación de estos moduladores consuma más tiempo que sus homólogos de tiempo discreto (debido al filtro de bucle analógico), ya que son especialmente insensibles a las imperfecciones del circuito y a las tolerancias de los componentes, disminuyendo así el ruido blanco generado. Por lo tanto, el modulador se diseñará de forma diferente a la habitual para que pueda ser implementado en una FPGA (Field Programmable Gate Array), reduciendo así el tiempo de simulación considerablemente en comparación a las simulaciones de simulink que se utilizan comunnmente, con el efecto añadido de que podrán simularse simultáneamente varios modelos en la FPGA sin que por ello disminuya la velocidad.

Nos pareció interesante reproducir por nuestra cuenta como abordaban este problema, debido a que simular moduladores Sigma-Delta de Tiempo Continuo es una tarea generalmente complicada y en la que se invierte demasiado tiempo en caso de que tengan que simularse varios moduladores al mismo tiempo. Y este enfoque de hardware programable ofrece una enorme aceleración en comparación con el ambiente simulink comúnmente utilizado.

1.2 Objetivos.

Como se señaló en el apartado anterior, este proyecto tiene como objetivo principal la implementación de un modelo hardware de simulación para moduladores Sigma-Delta de Tiempo Contínuo que permite acelerar el proceso de simulación de estos circuitos utilizando una FPGA.

Dicho objetivo se desglosará en los siguientes puntos:

- Estudio del método propuesto en el artículo de referencia [1].
- Diseño de un caso de estudio sobre el que aplicar el método del artículo. Para ello se diseñará un modelo en tiempo continuo de un modulador sigma-delta.
- Aplicación del método mencionado en el artículo.
- Implementar el modelo en una FPGA para comprobar el correcto funcionamiento del modelo. (Esto no ha sido realizado en este proyecto, debido a la longitud del mismo).

1.3 Estructura del documento.

En el Capítulo 1 "Introducción", se exponen los objetivos principales y la estructura del mismo.

En el capítulo 2 "Moduladores Sigma-Delta", se realizará una breve introducción a la evolución histórica de dichos moduladores, se introducirá un poco de teoría relacionada con los Moduladores Sigma-Delta de Tiempo Continuo y se explicará el estudio en el que ha sido basado este proyecto.

En el capítulo 3 "Diseño e implementacón", se realizará el diseño del modelo requerido y la explicación paso por paso de lo que se va haciendo.

En el capítulo 4 "Evaluación del diseño final", se mostrarán las simulaciones finales del sistema y se evaluará si este funciona o no correctamente tanto a través de MATLAB/Simulink como de ISE Desing Suite.

En el capítulo 5 “Presupuesto”, se recogerán los costes que supondrían la realización de este proyecto.

Las conclusiones a las que se han llegado y los posibles trabajos futuros se mostrarían en el capítulo 6, seguido de los anexos, en los que han sido recogidas las diferentes funciones de Matlab del proyecto, y la bibliografía utilizada.

2 MODULADORES SIGMA-DELTA.

2.1 Evolución histórica.

La modulación Sigma-Delta es un tipo de conversión digital/analógica o analógica/digital, que empezó a desarrollarse a principios de los años 50s, como una mejora de la modulación Delta, pero no fue hasta el año 1960 cuando el investigador C. C. Cutler patentó el primer circuito de este tipo, que consistía en un codificador de señal sobremuestreado que empleaba un bit de cuantificación. Posteriormente, H. Inose e Y. Yasuda publicaron un convertidor analógico/digital, en el año 1962, que denominaron modulador Delta- Sigma, el cual combinaba los principios de la modulación delta con un integrador [2]. Pero no fue hasta mediados de los 80s cuando empezaron a desarrollarse y a utilizarse ampliamente, gracias a la publicación de James C. Candy, quien introdujo la idea de una doble integración en el filtro de lazo del modulador [3]. Historicamente se afirma que Candy fue el precursor de los sistemas Sigma –Delta; no siendo él quien inventó este sistema de modulación, si no quien impulsó su desarrollo e investigación.

Gracias a los trabajos de Candy, en el año 1988 B. Bernhard publicó uno de los trabajos más relevantes en este campo, estableciendo así todas las consideraciones a tener en cuenta en el diseño de estos convertidores, algo muy novedoso para aquella época, que marcó el inicio de numerosas investigaciones, al igual que Candy.

Inicialmente, los convertidores Sigma- Delta se implementaron en el tiempo continuo, sin embargo, desde hace ya varias décadas se ha optado por implementaciones en el tiempo discreto, debido a la sencillez del trazado de las funciones matemáticas del modulador sobre el circuito. No obstante, esta implementación presenta limitaciones en el consumo de potencia y frecuencia de operación, obligando a retomar los covertidores continuos.

Durante aproximadamente 15 años a partir de la publicación de Candy, los convertidores Sigma – Delta se posicionaron como líderes en el procesamiento de voz y audio (tanto en la conversión del dominio analógico/digital, como el digital/analógico), instrumentación y sismografía.

Hoy en día, son altamente utilizados en comunicaciones, tanto cableadas como inalámbricas y, en general, cualquier aplicación que requiera de convertidores de alta resolución.

2.2 Moduladores Sigma-Delta de Tiempo Continuo.

La principal característica de un modulador Sigma-Delta es la obtención de una señal digital de gran resolución, utilizando un cuantificador de pocos niveles. El objetivo de estos convertidores es representar la señal de entrada con una secuencia de palabras digitales, cuyo espectro se aproxime a la señal analógica en una banda de interés, mientras que el restante sea solo ruido. Esto es posible al combinar técnicas como sobremuestreo, realimentación y procesamiento del error en un solo sistema, y así mejorar el desempeño del cuantificador. Idealmente, la resolución del modulador estará determinada por el orden del filtro, la relación de sobremuestreo y el número de niveles del cuantificador. Por tanto, si es posible incrementar la frecuencia de muestro, manteniendo lo demás, el ancho de banda del modulador se verá incrementado sin perder resolución.

Una forma de conseguir velocidades mayores en el reloj de muestreo consiste en implementar el filtro del modulador con circuitos en tiempo continuo (Figura 2.1), debido a que el ancho de banda de los dispositivos activos se puede reducir ya que operan con señales continuas y no discretas. Por ello son capaces de trabajar a mayores velocidades de reloj de muestreo respecto a sus homólogos de tiempo discreto, haciendo que, en teoría, un modulador Sigma-Delta de Tiempo Continuo pueda trabajar a una frecuencia de muestreo un orden de magnitud por encima sin perder resolución.

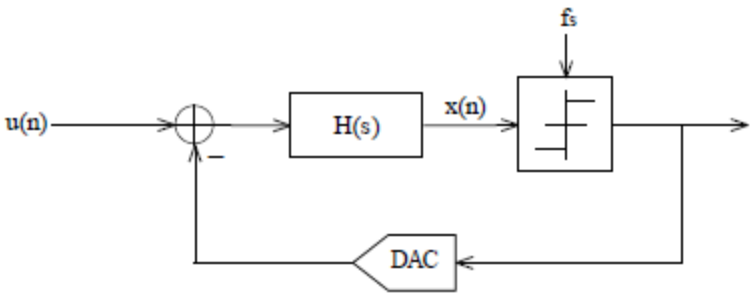

Figura 2.1 – Modulador Sigma-Delta en Tiempo Continuo.

Por otra parte, en los moduladores Sigma-Delta de Tiempo Continuo no aparecen *glitches* en los nodos de masa virtual de los amplificadores operacionales usados, lo que sí ocurre cuando empleamos circuitos con condensadores conmutados. Otro problema importante que presentan los moduladores Sigma-Delta de Tiempo Discreto es el fenómeno de *aliasing*, debido a que dos señales separadas por un múltiplo de la frecuencia de muestreo son indistinguibles. Por tanto, los moduladores Sigma-Delta de Tiempo Discreto necesitarán un filtro *antialiasing* a su entrada. Sin embargo, esto

puede evitarse en los moduladores Sigma-Delta de Tiempo Continuo debido a las características propias de filtrado que poseen.

A parte de las ventajas que los moduladores Sigma-Delta de Tiempo Continuo presentan frente a los moduladores Sigma-Delta de Tiempo Discreto, para su diseño tendremos que tener en cuenta una serie de problemas asociados a éstos. Como, los asociados a la incertidumbre de instante de muestreo y el retardo existente entre el instante de muestreo y el instante en que se actualiza la salida del DAC como consecuencia del muestreo. Aunque idealmente la corriente de salida del DAC responde inmediatamente al flanco del reloj de muestreo, en la práctica los transistores de los registros del cuantificador y el DAC no pueden conmutar instantáneamente. Este retardo que se origina en el camino de realimentación se denomina retardo de bucle en exceso o ELD (*Excess Loop Delay*). En los moduladores Sigma-Delta de Tiempo Discreto no existe un problema análogo por lo que este retardo es característico de los moduladores Sigma-Delta de Tiempo Continuo.

En cuanto al diseño, el método más comúnmente utilizado para diseñar el filtro del modulador Sigma-Delta de Tiempo Continuo es a partir de un filtro de Tiempo Discreto, que mostrará el comportamiento de acuerdo con las especificaciones de rendimiento. Sin embargo, nos encontramos con un problema, que no todos los filtros de moduladores de tiempo discreto se pueden realizar en el domino de tiempo continuo. Con la particularidad de que no hay pruebas de que un óptimo rendimiento en tiempo discreto también conduzca a un óptimo rendimiento en tiempo continuo. Por estos motivos y por las no linealidades o características únicas de los sistemas en tiempo continuo, su diseño no puede ser abordado a partir de un sistema de tiempo discreto en un proceso de diseño automatizado, como es el caso que se plantea en este proyecto.

Otra forma de diseñar moduladores Sigma-Delta de tiempo continuo es utilizar algoritmos de búsqueda heurística junto con simuladores analógicos como MATLAB/SIMULINK. Pero este método presenta una gran desventaja. Tal como mencionamos anteriormente, necesita un largo tiempo de simulación, derivado del muestreo del filtro analógico.

Por todos estos problemas, el método de elevación se va a realizar en una FPGA, basado en el entorno de simulación de moduladores Sigma-Delta de tiempo continuo.

Este enfoque de hardware programable provoca un aumento de la aceleración en tiempo de simulación, en comparación con el ambiente SIMULINK, normalmente utilizado, tal y como explicaremos en el siguiente apartado.

2.3 Aceleración de la simulación para moduladores Sigma- Delta de Tiempo Continuo [1].

En este apartado procederé a explicar el método que se ha intentado reproducir a lo largo de este proyecto, así como los resultados que obtuvieron Timon Brückner, Matthias Lorenz, Christoph Zorn, Joachim Becker, Wolgang Mathis y Maurits Ortmanns tras su aplicación, para poder tener una mejor comprensión del mismo y poder comparar, posteriormente, los resultados obtenidos en este proyecto con los que deberían haberse obtenido en realidad.

2.3.1 Diseño de moduladores Sigma-Delta de Tiempo Continuo.

Tal y como se ha mencionado con anterioridad, la simulación de moduladores de tiempo continuo consume mucho más tiempo que el de sus homólogos de tiempo discreto, provocando que procesos en los que se tengan que realizar un gran número de simulaciones simultáneas se demoren minutos, incluso horas. Aún así, los moduladores de tiempo continuo presentan numerosas ventajas frente a los de tiempo discreto, tal y como se ha explicado también en el apartado anterior, así que se desarrollará un método para que puedan ser implementados en una FPGA, reduciendo así el tiempo de simulación en un factor de 10^5 en comparación a una simulación de Simulink, lo que permitirá la simulación de 10000 moduladores por segundo.

Como se comentó en el Apartado 0, el método más comúnmente utilizado para realizar el diseño de moduladores Sigma-Delta de Tiempo Continuo sería diseñar el filtro a partir de un filtro de Tiempo Discreto, pero debido a los problemas que este método conlleva, se vió descartado. Al igual que el de los algoritmos de búsqueda heurística junto con simuladores analógicos como MATLAB/SIMULINK, debido al largo tiempo de simulación que se necesitaría, por lo tanto se utilizará un método que permitiría simular con precisión un modulador Sigma-Delta en el dominio de Tiempo Discreto mediante la corrección de los estados internos en cada caso de muestreo. La naturaleza de tiempo discreto de este método es muy adecuada para la externalización en un hardware ya que así se provocará una aceleración en un factor de $1.2*10^5$ del tiempo de simulación, en comparación con el ambiente de Simulink utilizado normalmente. Solucionándose así los problemas de simulación en el diseño de moduladores Sigma-Delta de Tiempo Continuo.

2.3.1.1 Descripción del método.

Para señales de entrada conocidas, los moduladores Sigma-Delta de Tiempo Continuo se pueden simular con precisión en el dominio de Tiempo Discreto. Por lo que se diseñará un filtro de Tiempo Continuo, que posteriormente será discretizado mediante una transformación Tiempo Continuo a Tiempo Discreto con retención de orden cero (ZOH). Además, las diferencias entre el modelo de tiempo continuo y su discretización se calcularán para cada paso de la simulación, añadiéndose esta a los estados internos, x(z), del filtro de Tiempo Discreto en cada caso de muestreo, para corregir las desviaciones de la discretización del modelo de Tiempo Continuo.

2.3.1.1.1 Modelo en Tiempo Continuo.

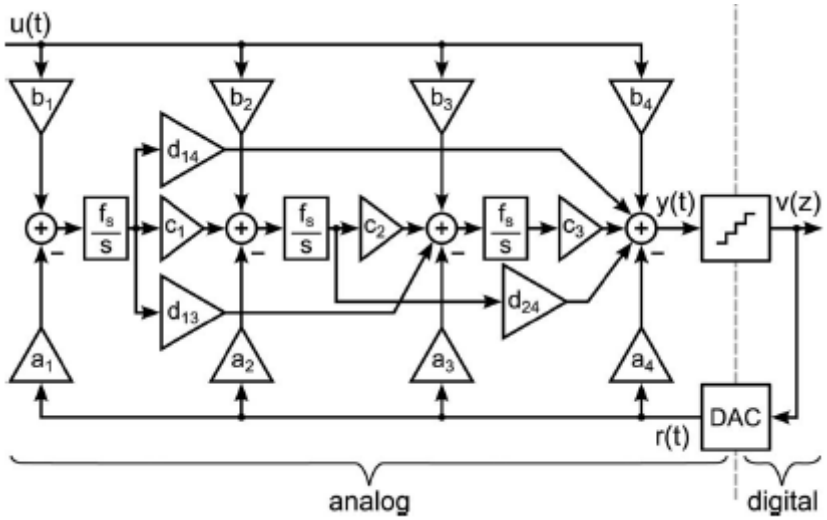

Figura 2.2- Modulador Sigma-Delta de tiempo continuo con un filtro de tercer bucle genérico (diseño de partida).

En primer lugar será necesario diseñar un modelo en Tiempo Continuo para su posterior discretización, tal y como se ha explicado con anteriroridad. Para ello se utiliza como diseño de partida un filtro de tercer bucle genérico tal como se muestra en la Figura 2.2. Como puede observarse, el filtro consta de una cadena de integradores de Tiempo Continuo ideales, con todas las conexiones posibles excepto resonadores locales. Donde la salida del cuantificador se utiliza para realimentar el filtro, realizando una conversión mediante un convertidor digital-analógico (DAC), que puede llevarse a cabo sin retardo o con un retardo máximo de un ciclo horario (ELD), en la configuración propuesta.

Posteriormente se elegirán los valores adecuados para los coeficientes del filtro, generando así las matrices de espacio de estados que se normalizan a la frecuencia de muestreo f_s y son dadas por (1).

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)_{CT} = \left(\begin{array}{ccc|cc} 0 & 0 & 0 & b1 & -a1 \\ c1 & 0 & 0 & b2 & -a2 \\ d13 & c2 & 0 & b3 & -a3 \\ \hline d14 & d24 & c3 & b4 & -a4 \end{array}\right)_{CT} \quad (1)$$

2.3.1.1.2 Discretización.

Una vez diseñado el modulador en Tiempo Continuo, habrá que realizar la discretización de éste. Comenzando por la transformación de los integradores del filtro, la cual se realiza por etapas, tal y como se muestra en la Figura 2.3, donde el comportamiento de dichos integradores no se verá alterado a pesar de sus diferencias.

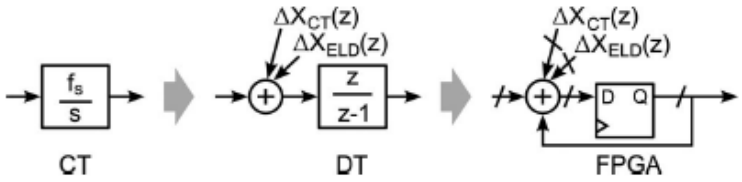

Figura 2.3 - Emulación de un integrador analógico de tiempo continuo en hardware digital

En cuanto a la transformación ZOH, ésta se lleva a cabo a partir de la teoría de control, de acuerdo con (2), (3), (4) y (5). Donde A_{CT}, B_{CT}, C_{CT} y D_{CT} son las matrices de espacio de estados de Tiempo Continuo (1). Tras dicha transformación, el espacio de estados final del modulador se convierte en (6), pudiendo observarse que éste ha sido ampliado. La ampliación consiste en dos términos de corrección de entrada adicionales por estado interno, que son los términos de corrección de errores: uno de ellos será el que restaure el comportamiento del circuito debido a la discretización del modelo en tiempo continuo y el otro se utilizará para tener en cuenta la modulación de ELD (retardo de realimentación).

$$A_{DT,ZOH} = e^{A_{CT}} \quad (2)$$

$$B_{DT,ZOH} = \int_0^1 e^{A_{CT}(1-\xi)} B_{CT} d\xi \quad (3)$$

$$C_{DT,ZOH} = C_{CT} \quad (4)$$

$$D_{DT,ZOH} = D_{CT} \quad (5)$$

$$\left(\begin{array}{ccc|cccccccc} 1 & 0 & 0 & b1 & -a1 & 1 & 0 & 0 & 1 & 0 & 1 \\ c1 & 1 & 0 & b2 & -a2 & 0 & 1 & 0 & 0 & 1 & 0 \\ d13 & c2 & 1 & b3 & -a3 & 0 & 0 & 1 & 0 & 0 & 1 \\ \hline d14 & d24 & c3 & b4 & -a4 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) \quad (6)$$

Quedando un vector de entrada resultante (7) que contiene las entradas originales y los términos de corrección, $\boldsymbol{\Delta X_{CT}}$ y $\boldsymbol{\Delta X_{ELD}}$. Donde la señal de entrada ***U(t)*** es un vector de la entrada del modulador ***u(t)*** y ***r(t)*** la señal de realimentación cuantificada.

$$U(z) = [u(z)\, r(z) \;\; \Delta \mathrm{X}_{CT}(z)^T \;\; \Delta \mathrm{X}_{ELD}(z)^T]^T \quad (7)$$

En el caso de estudio, en el que se utiliza una única señal de entrada de ondas sinusoidal, los términos de corrección debidos a la discretización formarán ondas sinusoidales individuales de la misma frecuencia pero con diferentes amplitudes y fases, como se muestra en (8). Y para señales de entrada constantes en la presencia de ELD los términos de corrección serán valores constantes, multiplicado por la diferencia de las dos últimas salidas muestreadas, tal como puede verse en (9). Por lo tanto, para las señales de entrada dadas, todos los términos de corrección podrán ser precalculados para cualquier filtro de Tiempo Continuo.

$$\Delta \mathrm{X}_{\mathrm{CT}}(\mathrm{z}) = \int_0^1 e^{A_{CT}(1-\xi)} B_{CT}\{U(z+\xi) - U(z)\} d\xi \quad (8)$$

$$\Delta \mathrm{X}_{\mathrm{ELD}}(\mathrm{z}) = \int_0^{ELD} e^{A_{CT}(1-\xi)} B_{CT}\{r(z-1) - r(z)\} d\xi \quad (9)$$

2.3.1.1.3 Representación de la estructura en hardware.

En cuanto a la represetación digital, ésta se realiza mediante la elección de una longitud de bits apropiada para todas las señales y la sustitución de los integradores de Tiempo Discreto por un registro de un bucle de realimentación. En una simulación secuencial de MATLAB, esta precisión normalmente es de 64 bits (número representado en coma flotante), pero hay que tener en cuenta que el rendimiento del circuito está generalmente limitado por el ruido térmico de entrada, por lo que la longitud de bits de las señales de entrada se deberá ajustar a un valor apropiado con el fin de limitar la influencia del ruido térmico esperado (tomándose una resolución entre 10 y 16 bits).

Además de las etapas del integrador, la estructura del integrador tiene que estar también representada en hardware. Para ello hay que tener en cuenta que todos los

coeficientes del filtro distintos de cero en (6) corresponden a una multiplicación y una adicción, que van a ser ejecutadas de forma simultánea durante cada instancia de muestreo. En la FPGA dichas multiplicaciones pueden realizarse en paralelo, en un ciclo de reloj, mediante multiplicadores cableados, y las sumas en la entrada de cada integrador y en frente del cuantificador interno se realizan en árboles binarios de adicción, para explotar así la capacidad de cálculo del hardware. Destacando que el cuantificador interno tiene una resolución de 1 a 7 bits, la salida se almacenará termporalmente en la RAM de Bloque (Brams) para su posterior procesamiento. Por ultimo, para formar el bucle del modulador, la salida realimentará de nuevo en el filtro, que actúa como DAC NRZ. Pudiendo realizarse formas de onda del CAD más complicadas mediante extensiones de los términos de elevación.

Finalmente, después de todos los cambios realizados desde el diseño de partida, se obtiene un modelo como el que aparece en la Figura 2.4, donde el cuantificador realiza una transformación de la precisión de la señal interna con una determinada longitud de bits de cuantificación elegidos, mientras que el DAC hace lo contrario con el fin de formar el bucle de realimentación. Por otra parte, la última ganancia -a4 de realimentación se retrasará, con el fin de evitar un sistema no causal. Lo cual no cambiará el comportamiento del modulador, debido a la fase de mantenimiento en el cuantificador.

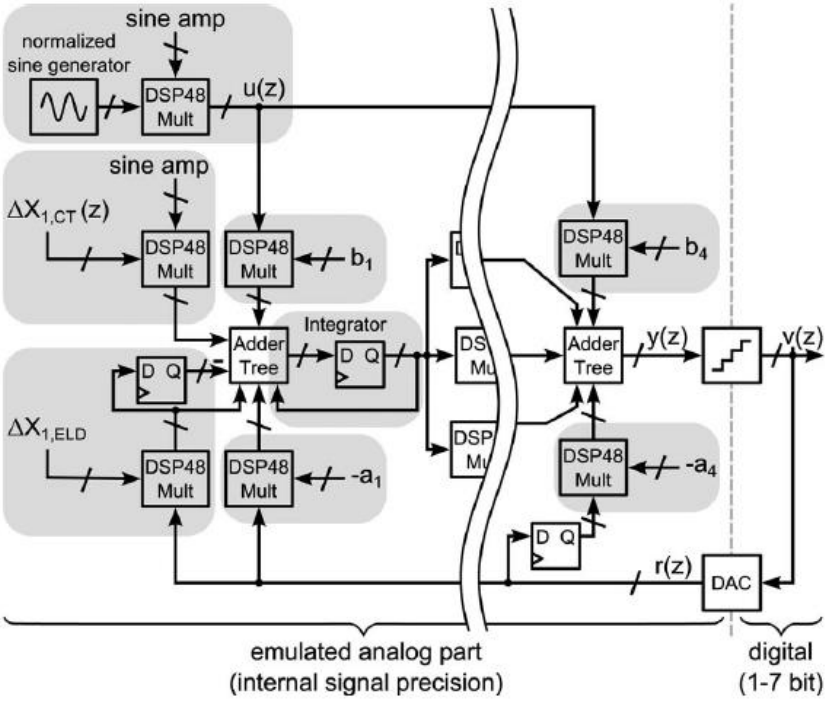

Figura 2.4 – Implementación del modulo genérico del modulador Sigma-Delta.

2.3.2 Evaluación de la FPGA.

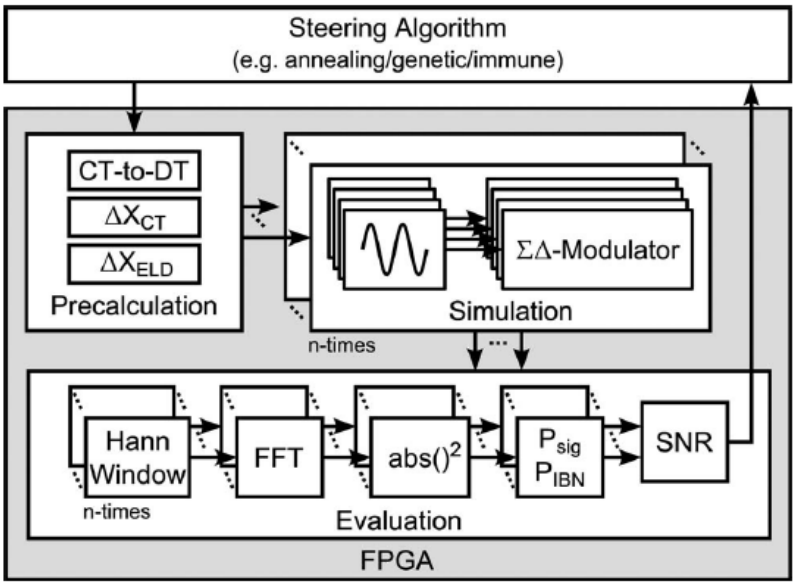

Figura 2.5 - Entorno de simulación y evaluación con n módulos de simulación que realizan 4n simulaciones paralelas en la FPGA.

En la Figura 2.5 se muestra el entorno de evaluación de alta velocidad implementado para moduladores Sigma-Delta de Tiempo Continuo que se ha desarrollado, que pueden utilizarse para, por ejemplo, algoritmos heurísticos de diseño. Con el fin de escalar con el tamaño de diferentes FPGAs, el módulo de simulación puede ser duplicado n veces. Además, cada módulo de simulación consta de cuatro submódulos de emulación paralelos (modulador ΣΔ) con el fin de utilizar la capacidad completa de la velocidad de la realización de la transformada rápida de Fourier (Fast Fourier Transform, FFT).

En el medio mostrado se ha generado la mayor flexibilidad posible para permitir el uso de cualquier algoritmo de dirección que requiera simulaciones masivas. Por lo tanto, es posible seleccionar una frecuencia individual y la amplitud de la señal de entrada, la relación de sobremuestreo, una estructura específica de filtro de acuerdo con la Figura 2.2, un ancho de bits entre 1 y 7, y un valor para ELD entre cero y una veces el ciclo de trabajo del reloj de muestreo para cada modulador solo bajo prueba.

2.3.2.1 Cálculos iniciales.

La transformación de acuerdo con (2), (3), (4) y (5), puede realizarse de forma bastante más compacta en la FPGA debido a las cortas ecuaciones analíticas. Esto también es cierto para los términos de corrección para ELD, al moldear formas de onda rectangularesdel CAD. Por el contrario, los cálulos previos de los términos de corrección de entrada de (8) son bastante complicados, y almacenar todos los valores calculados para cada paso de tiempo solo daría lugar a una gran demanda de memoria. Por lo tanto, se toma que, para las entradas individuales de onda senosoidal, los términos de corrección relacionados $\Delta X_{CT}(z)$ formen de nuevo las ondas sinusoidales individuales de la misma frecuencia, pero de diferentes amplitudes y fases. Para el cálculo de estos valores, se requiere una función de raíz cuadrada de lo absoluto y una función tangente inversa para la fase. Mientras que la función de raíz cuadrada podría ser generada por el XILINX CORE GENERATOR, la tangente inversa podrá ser realizada mediante una tabla de búsqueda.

2.3.2.2 Generación de señal.

La señal de entrada y los términos de corrección relacionados son generados por los generadores de ondas sinusoidales múltiples. Donde cada generador emite las cuatro ondas sinusoidales solicitadas de la misma frecuencia para un modulador de tercer orden. Para evitar la fuga de señal, la frecuencia será elegida a un bin exacto de la FFT. Y los armónicos inherentes de los generadores de onda senosoidal se ajustan en función de la anchura de bits de su representación de la señal y se reducen al aumentar el ancho de bits hasta un punto en que el resultado final no se vea comprometido.

2.3.2.3 Evaluación.

Para realizar la evaluación de la señal de salida de la emulación de los moduladores, se implementa una cadena de proceso común, mostrada en la parte inferior de la Figura 2.5

1) Para surimir las fugas de ruido, el flujo de bits de salida se selecciona mediante una ventana de Hann y es remitido a la FFT (transformada rápida de Fourier). La secuencia de entrada FFT contendrá 2^{14} muestras, lo que limitará, teóricamente, la relación de sobremuestreo a valores razonables de unos pocos cientos. (El núcleo FFT habrá sido generado por el generador de CORE y ejecuta una transformación de tiempo a frecuencia).

2) En el módulo siguiente se calculan los valores absolutos cuadrados de las salidas FFT complejas.
3) Posteriormente, la potencia de riudo en banda, P_{IBN}, se calcula hasta el borde de banda deseado y los contenedores de frecuencia que contengan la señal se acumularán por separado en la Psig. La SNR (relación señal a ruido) se obtendrá de la diferencia de Psig y P_{IBN} después de calcular un logaritmo de búsqueda basado en tablas para evitar una división de alta resolución.
4) Finalmente, los valores de SNR calculados, se almacenarán en un BRAM y serán devueltos al algoritmo de dirección.

A diferencia de la emulación, que no puede ser pipeline de manera eficiente, debido al bucle de realimentación, la evaluación es completamente pipeline. De este modo, solo se necesita un módulo de SNR para un gran número de caminos de evaluación, mientras que cada camino es capaz de manejar cuatro salidas de emulador, desplazadas en el tiempo, tal como se muestra en la Figura 2.5.

2.3.2.4 Requisitos de hardware.

El diseño es escalable, de modo que, dependiendo de las limitaciones de hardware, pueden ser instanciados más módulos de simulación con los correspondientes caminos de evaluación. En FPGAs más grandes, se podrán hacer más simulaciones, por lo tanto estas se podrán ejecutar en paralelo, aumentando el número alcanzable de simulaciones por segundo, lo que acelerará, por ejemplo, aún más las búsquedas heurísticas. En la Tabla 1 se enumeran los requisitos de hardware para un diseño que está dimensionado para llenar todo el chip de prueba XILINX XCV6VLX24OT-1. En esta moderna pero pequeña FPGA se pueden crear instancias en paralelo de cuatro módulos de simulación con 16 bits de la señal de punto fijo de precisión. Por lo tanto, se podrán investigar 16 moduladores Sigma-Delta simultáneamente.

		Slices	DSP48	18kB BRAM
Initial calculations	(1x)	7142	67	60
Simulation module	(4x)	1964	88	72
Sine generators	(4x)	158	-	10
ΣΔ modulators	(4x)	333	22	8
Evaluation module	(1x)	10946	258	280
FFT core	(4x)	2685	64	41
SNR core	(1x)	206	2	116
Total environment		25944	677	628
Chip usage (XCV6VLX240T-1)		69%	88%	75%

Tabla 1 - Recursos requeridos en un VIRTEX-6. 16 moduladores Sigma-Delta paralelos con exactitud de 16-bits.

2.3.3 Resultados y discusion.

Con el fin de probar la operatividad y factibilidad del enfoque de aceleración de hardware propuesto, el entorno de simulación del modulador Sigma-Delta de Tiempo Continuo introducido, se analiza en términos de precisión y velocidad.

2.3.3.1 Análisis de precisión.

Los resultados de la simulación se muestran a modo de ejemplo para un modulador de tercer orden, con retroalimentación compensada, un cuantificador de 4 bits y 50% ELD.

Como puede observarse en la Figura 2.6, el número de decimales en la representación de punto fijo de las señales en el diseño influye directamente en la resolución espectral. Este efecto es estudiado para precisiones de señales de 10 bits, 13 bits y 16 bits dentro del módulo emulador. Donde se realiza la ventana de Hanns y la FFT con 16 bits de precisión de entrada de punto fijo en los tres casos. Limitando la resolución de la señal resultante a un ruido de fondo correspondiente, que limita la SNR alcanzable. A pesar de que es mucho más bajo, también la referencia de MATLAB exhibe un nivel de ruido de truncamiento de la señal, debido a la precisión de 64 bits de punto flotante.

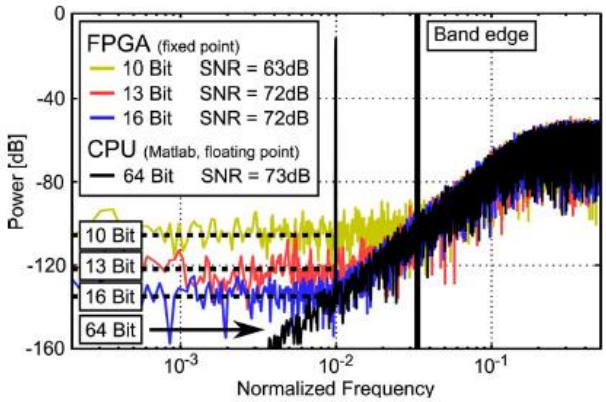

Figura 2.6 - Salida FFT de la FPGA para diferentes anchos de bit de señal.

Tal y como se muestra en la Figura 2.6, la resolución de 10 bits no es suficiente para la evaluación del modulador, ya que el ruido de banda dominará el comportamiento. Sin embargo una resolución de 13 bits, es suficiente para obtener el valor SNR correcto para este modulador.

Por último, para comprobar que se obtiene lo mismo realizándose la simulación a través de MATLAB o a través de la FPGA, se representa una SNR de 60 diferentes amplitudes, Figura 2.7, pudiendo verse que las curvas serán perfectamente coincidentes (dentro de la precisión de este método de evaluación).

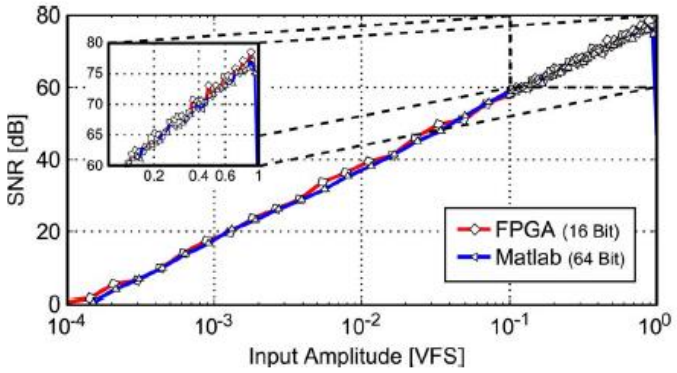

Figura 2.7 - Comparación de SNR alcanzado por FPGA y simulaciones de Matlab.

2.3.3.2 Análisis de velocidad.

	single run	10^6 runs
CPU[2] (ode45, Dormand-Prince) Matlab® Simulink®	2.42 s	28 d
CPU[2] (lifting) Matlab®	10.73 ms	3 h
FPGA[3] (lifting, on-chip) 16 parallel simulations	20.88 μs	21 s

[1] FFT length of 2^{14} [2] Intel Core i7 950 [3] XCV6VLX240T-1

Tabla 2 - Comparación de simulación y evaluación de tiempos.

La velocidad alcanzable en el sistema propuesto se compara con una simulación SIMULINK basada en el CPU común de la Tabla 2. Pudiendo comprobar que el tiempo de procesamiento FPGA puro para una única simulación es 20.88µs, lo que resulta de 47893 simulaciones por segundo. En comparación con Simulink, esto produce un aumento de velocidad en un factor de $1.2*10^5$ para la FPGA utilizada.

2.3.4 Conclusión.

Después del o explicado anteriormente, se puede concluir que en un entorno de simulación basado en una FPGA para moduladores Sigma-Delta de Tiempo Continuo las simulaciones se acelerarán en un factor de más de 10^5 en comparación con simulaciones en una configuración de SIMULINK comúnmente utilizadas.

3 DISEÑO E IMPLEMENTACIÓN

Una vez explicado a lo que se quiere llegar con este proyecto, procederemos a realizar nuestro propio diseño para comprobar que realmente se obtiene una aceleración en la simulación de moduladores Sigma-Delta de Tiempo Continuo.

Tal y como se mencionó en el apartado anterior, para señales de entrada conocidas, como es el caso de este proyecto, los moduladores Sigma-Delta de Tiempo Continuo se pueden simular con precisión en el dominio de tiempo discreto. Por lo tanto el filtro de tiempo continuo aquí descrito, será discretizado mediante una transformación tiempo continuo a tiempo discreto con retención de orden cero (ZOH) y las diferencias entre el modelo continuo y las ecuaciones lineales estacionarias discretizadas se calcularán para cada paso de la simulación. Esta diferencia será añadida a los estados internos del filtro en tiempo discreto en cada caso de muestreo y el modelo digital equivalente que obtengamos se implementará en ISE.

Como diseño de partida se utilizará el modulador Sigma-Delta de Tiempo Continuo con filtro de tercer orden (diagrama de bloques de la Figura 2.2), sobre el que habrá que realizar diversas modificaciones a lo largo de las diferentes fases del diseño, para la obtención del modelo final.

3.1 Diseño en tiempo continuo.

Como puede observarse en la Figura 2.2, el filtro estará formado por una cadena de integradores de tiempo continuo ideales con todas las conexiones posibles, a excepción de los resonadores locales, que se han omitido para simplificar el modelo.

Ahora será necesario elegir los valores adecuados para los coeficientes del filtro, las características de la señal de entrada, el diseño del cuantificador a utilizar, y el DAC (que formará parte de la realimentación).

3.1.1 Coeficientes del Filtro.

Estos se obtienen a través de la generación de unas matrices de espacio de estado, que se normalizan a partir de una frecuencia de muestreo seleccionada meticulosamente y son dadas por (1), a través de una determinada función de MATLAB (

ANEXO 4 – FUNCIONES NECESARIAS.), que ha sido desarrollada previamente.

La tasa o frecuencia de muestreo es el número de muestras por unidad de tiempo que se toman de una señal continua para producir una señal discreta, durante el proceso necesario para convertir esta de analógica a digital. En el caso de estudio ésta fue tomada como $50 \cdot 10^6$ Hz.

Es necesario destacar que las matrices de espacio de estados obtenidas mediante Matlab no son iguales a las que aparecen en (1), si no que devuelve la matriz mostrada en (10), en la que los coeficientes ***a*** se han visto separados en diferentes columnas y sólo aparece el valor del coeficiente ***b1***. Aunque esto no supone un gran problema, ya que los coeficientes ***b*** restantes son iguales a los ***a***, solo que de signo contrario. Haciendo que la reorganización de los datos no sea una tarea demasiado complicada.

$$\left(\begin{array}{ccc|ccccc} 0 & 0 & 0 & 0.1617 & -0.1617 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & -0.6609 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1.3237 & 0 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 & -0.5826 \end{array}\right) \quad (10)$$

Posteriormente los coeficientes pueden guardarse en arrays para poder manejarlos con mayor facilidad, siendo estos los que aparecen en (11), (12), (13), (14). Donde d(1) corresponde a ***d13***, d(2) a ***d14*** y d(3) a ***d24***.

$$a = (-0.1617 \quad -0.6609 \quad -1.3237 \quad -0.5826) \quad (11)$$

$$b = (0.1617 \quad 0.6609 \quad 1.3237 \quad 0.5826) \quad (12)$$

$$c = (1 \quad 1 \quad 1) \quad (13)$$

$$d = (0 \quad 0 \quad 0) \quad (14)$$

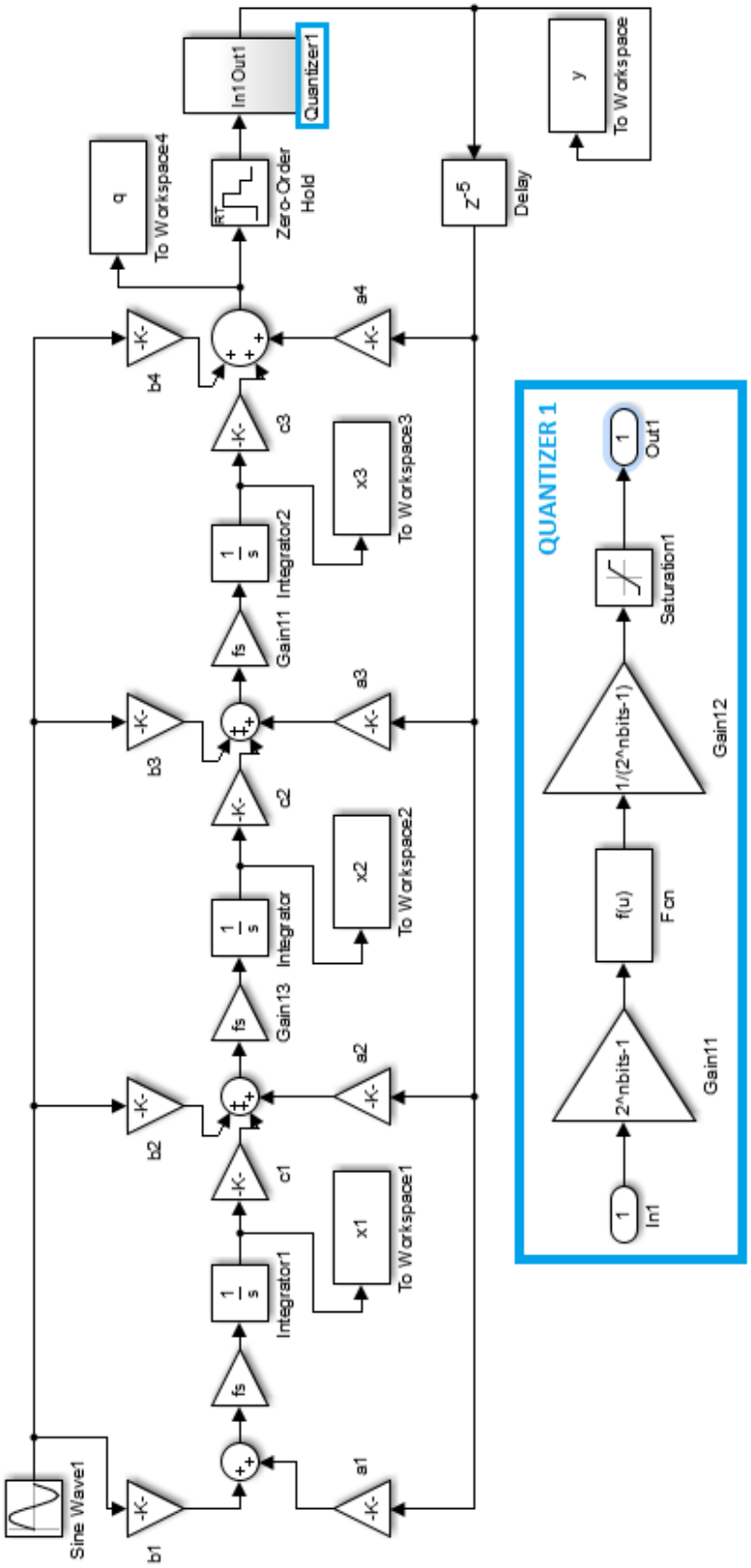

Figura 3.1 – Modelo en tiempo continuo.

3.1.2 Cuantificador.

El cuantificador es un elemento del sistema que es el encargado de realizar una transformación de la precisión de la señal interna en el ancho de bits de cuantificación. En el caso de estudio, el ancho de bits de cuantificación será de tres.

Hay que destacar que como la señal que recibiría el cuantificador, en este caso, no estaría previamente muestreada, al encontrarnos en un caso de tiempo continuo, es necesario que el cuantificador esté precedido de un Zero-Holder Hold, como aparece en la Figura 3.1, que se encargue de muestrear la señal durante un tiempo de muestreo determinado. Es decir, el Zero-Holder Hold toma valores de la señal recibida cada vez que transcurre un determinado periodo de muestreo (15). Donde f_s es la frecuencia de muestreo, tomándose ésta como $50 \cdot 10^6$Hz, tal y como se dijo en el apartado anterior, pudiendo ser modificada en el caso de que el modelo no funcionase correctamente. También es necesario seleccionar el tiempo de simulación, ***tsim***, que se calcula mediante (16), donde ***lsim*** es la longitud de simulación. Se conoce como longitud de simulación el número de puntos que se quieren muestrear para realizar la posterior discretización. En el caso de estudio se ha establecido una longitud de 13 bits, ya que por ahora no es necesaria una gran cantidad de puntos de muestreo, por lo tanto la longitud de simulación será de 2^{13}.

$$T_s = {}^{1}/_{f_s} \tag{15}$$

$$tsim = lsim \cdot T_s \tag{16}$$

3.1.3 Señal de entrada.

La señal de entrada elegida ha sido una onda sinusoidal de las características que se explican a continuación, debido a que los cálculos en los que participa para la posterior discretización del modelo serán más fáciles de realizar que si se utilizase otro tipo de señal, como una señal cuadrada o triangular, por ejemplo. Esto será explicado más ampliamente en el apartado 0.

Se procede entonces a elegir la amplitud y la frecuencia. En el caso de estudio se tomó el valor de amplitud, ***ampin,*** de -10 dB y una frecuencia de entrada, ***fin***, considerada, en este caso, como un 30% del ancho de banda (BW).

3.1.4 DAC.

Como DAC se utiliza un bloque *Delay*, tal como se muestra en la Figura 3.1, que será el encargado de retrasar la señal de salida un periodo de tiempo determinado para que se produzca una correcta realimentación. Es decir, deshará el trabajo del cuantificador. En el caso de estudio ha sido seleccionado un periodo de Ts/10, que no ha sido necesario modificar, ya que ha sido obtenida una snr satisfactoria (cosa que estudiaremos más adelante).

3.1.5 Evaluación y conclusión.

Una vez completado el diagrama de bloques del conversor Sigma-Delta de Tiempo Continuo (Figura 3.1), es cuando se deberá simular el modelo para proceder a un exhaustivo análisis en el que se comprobará el correcto funcionamiento del sistema.

En primer lugar se analizan las variables de estado del sistema, representadas en la Figura 3.2, ya que son el primer indicador de que el sistema funciona o no correctamente, siendo estas ***q***, ***x1***, ***x2*** y ***x3***.

A continuación, se procede a estudiar la señal de salida ***y***, representada en la Figura 3.3, que muestra que la señal seno de entrada ha sido transformada en una señal digitalizada de tres bits, que se repite periódicamente en el tiempo. Siendo este el resultado buscado.

Pero, a pesar de todo, aunque las comprobaciones anteriores hayan sido satisfactorias, no son suficientes para admitir que el diseño es el adecuado. Teniendo que analizar el espectro de frecuencias de la Figura 3.4, que ha sido representado para asegurar que la máxima densidad espectral se encuentra aproximadamente a la misma frecuencia que la de la señal de entrada, ***fin***, siendo esta de 232 kHz. Y para comprobar que a partir de límite superior del ancho de banda, representado por la línea roja, es cuando aparece el ruido. Ambas cosas se cumplen, dando una nueva razón para pensar que el diseño ha resultado satisfactorio.

Para realizar la última comprobación, sería necesario conocer la SNR (relación señal a ruido) ideal que debería tener el sistema, para ello se desarrolló una función en MATLAB que simula un modulador sigma-delta de tiempo continuo con entrada de ondas sinusoidales de diferentes amplitudes, que calcula la relación señal a ruido para cada entrada. Obteniéndose así una SNR objetivo de valor 86dB, que es nuestra referencia a lo largo del proyecto para comprobar que el diseño se está realizando de

forma satisfactoria. Dicho esto, procedemos a estudiar la SNR de nuestro sistema. Este valor se calcula a partir de la señal de salida ***y***, obteniéndose un valor de 76.0984 dB, que es bastante bueno, teniendo en cuenta el tipo de modulador que estamos estudiando. Concluyendo de esta forma, que el diseño del Modulador Sigma-Delta de Tiempo Continuo realizado es válido y que puede procederse a la siguiente fase de diseño.

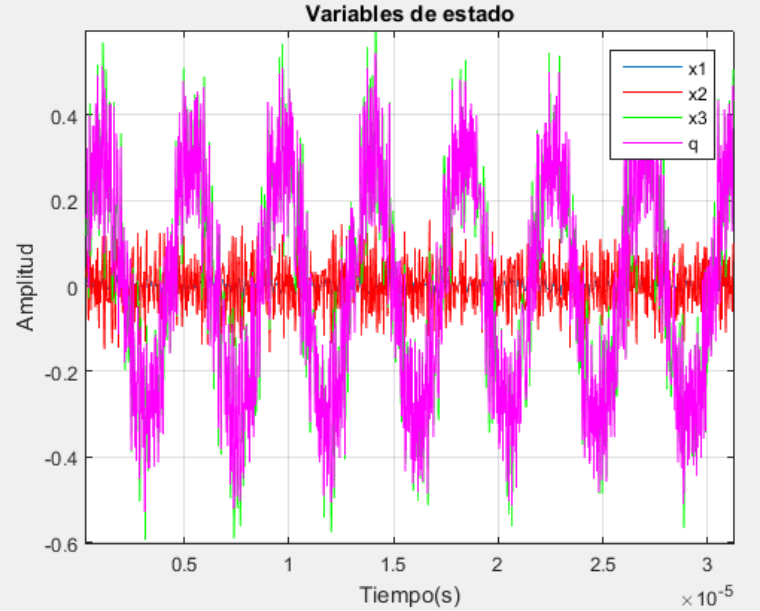

Figura 3.2 – Variables de estado (q, x1, x2 y x3) en función del tiempo (modulador Sigma-Delta de Tiempo Continuo).

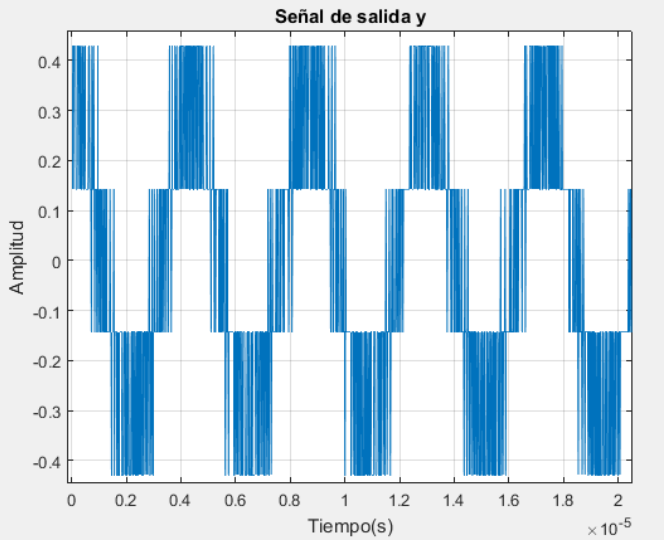

Figura 3.3 – Señal de salida 'y' en función del tiempo (modulador Sigma-Delta de Tiempo Continuo).

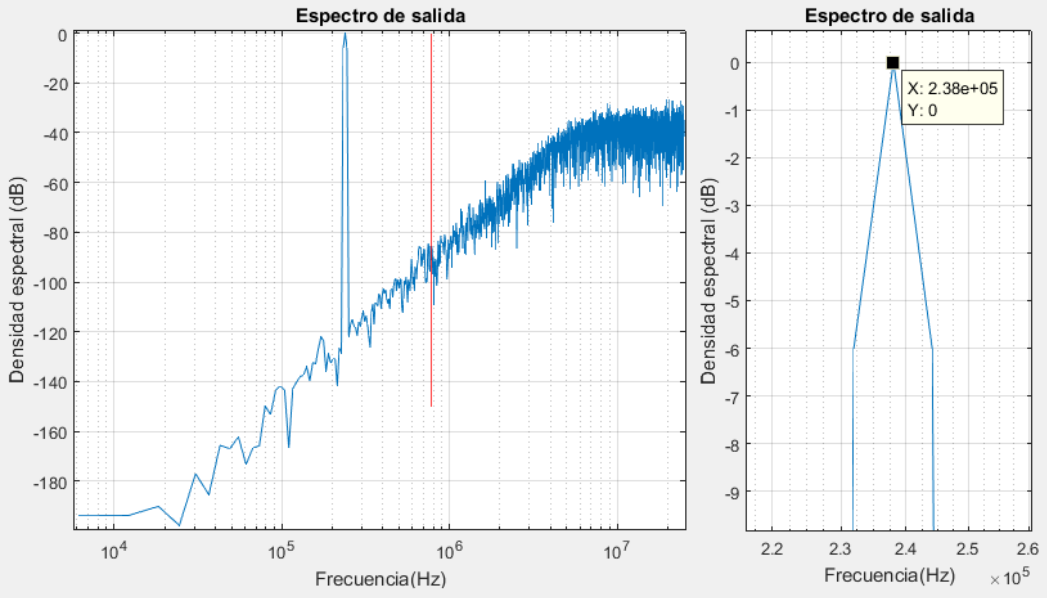

Figura 3.4 - Espectro de salida del modulador (modulador Sigma-Delta de Tiempo Continuo).

3.2 Diseño en tiempo discreto.

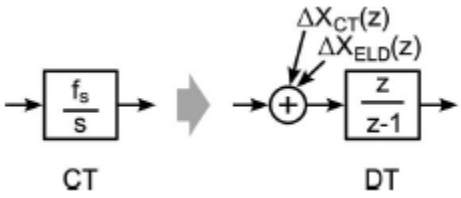

Figura 3.5 - Etapas del integrador para la transformación a tiempo discreto.

Una vez realizado el diseño del modelo en tiempo continuo, es necesario discretizarlo. Para ello, se realiza una transformación de orden cero (ZOH) y una ampliación la matriz de espacio de estados final del modulador, tal como se explicó en el Apartado 2.3.1, inluyendo así los llamados términos de corrección, $\boldsymbol{\Delta X_{CT}}$ y $\boldsymbol{\Delta X_{ELD}}$.

Aparte de lo anteriormente descrito se realizará una transformación de los integradores, tal y como aparece en la Figura 3.5.

3.2.1 Transformación con retención de orden cero (ZOH).

La transformación con retención de orden cero se realiza mediante (2), (3), (4) y (5). Obteniéndose así, también, los nuevos coeficientes necesarios para el diseño, mostrados en (17), (18), (19) y (20), donde d(1) corresponde, nuevamente, a ***d13***, d(2) a ***d14*** y d(3) a ***d24***.

$$a = (-0.1617 \quad -0.7417 \quad -1.6811 \quad -0.5826) \tag{17}$$

$$b = (0.1617 \quad 0.7417 \quad 1.6811 \quad 0.5826) \tag{18}$$

$$c = (1 \quad 1 \quad 1) \tag{19}$$

$$d = (0.5 \quad 0 \quad 0) \tag{20}$$

3.2.2 Cálculo de los términos de corrección.

Como ha sido explicado en el Apartado 2.3.1.1.2, los términos de corrección debidos al ELD pueden ser calculados con facilididad. Teniendo en cuenta que únicamente se tomarán los valores de la segunda columna de la matriz $\boldsymbol{B_{CT}}$, que son los correspondientes a los coeficientes de la rama de realimentación del filtro, quedarán unos términos de corrección debido al ELD como el que aparece mostrado en la Figura 3.6. Donde ***lif_eld*** será cada uno de los coeficientes del vector, correspondiente a cada integrador. Es decir, que en el filtro de Tiempo Discreto se introducirá un término de corrección debido al ELD por cada uno de los integradores, tal y como se muestra en la Figura 3.7.

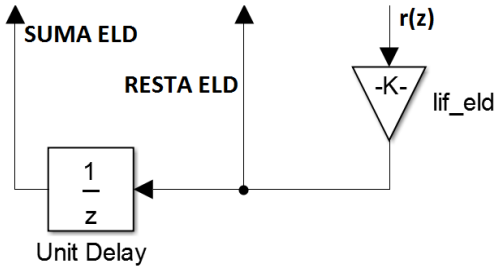

Figura 3.6 – Término de corrección $\Delta X_{ELD}(z)$.

En cuanto a los términos de corrección debidos a la discretización, su cálculo anticipado será bastante más complicado que en el caso anterior. Ya que se necesitaría una gran demanda de memoria para almacenar todos los valores calculados para cada paso de tiempo. Para solucionar este problema se hace uso del hecho de que los términos de corrección de entrada deberán volver a formar de nuevo ondas sinusoidales individuales de la misma frecuencia que la señal de entrada u(t), pero de diferentes amplitudes y fases. En el estudio realizado por Timon Brückner, Matthias Lorenz, Christoph Zorn, Joachim Becker, Wolgang Mathis y Maurits Ortmanns [1], se propone la utilización de una función raíz cuadrada de valor absoluto y una función tangente inversa para la fase. Sin embargo esto también resulta complicado, así que se decidió utilizar (21), tomado de un nuevo estudio publicado por Timon Brückner, Martin Kiebler, Matthias Lorenz, Christoph Zorn, Jens Anders, Wolgang Mathis y Maurits Ortmanns [4]. Donde $\boldsymbol{A_{in}}$ será considerada como la amplitud de la señal sinusoidal de entrada, $\boldsymbol{\omega}$ la frecuencia de la señal sinusoidal de entrada expresada en radianes, $\boldsymbol{A}$ la matriz $\boldsymbol{A_{CT}}$, $\boldsymbol{B_{in}}$ la matriz $\boldsymbol{B_{CT}}$ (de la cual solo se tomarán los valores de la primera columna, que corresponden con los coeficienes de la rama de entrada) y $\boldsymbol{\varphi}$ tendrá valor nulo. Resolviendose así el problema de cálculo de los términos de corrección de entrada, que serán introducidos en el filtro de tiempo discreto tal y

como se muesta en la Figura 3.7.

$$\Delta X[n+1] = A_{in}\sin(\omega n+\phi)\,Re\left\{\int_0^1 e^{A(1-\xi)}B_{in}e^{j\omega\xi}d\xi\right\} \tag{21}$$

$$+A_{in}\sin(\omega n+\phi)\,Im\left\{\int_0^1 e^{A(1-\xi)}B_{in}e^{j\omega\xi}d\xi\right\}$$

3.2.3 Cuantificador.

El cuantificador, en este caso, realizará la misma función que en el diseño en tiempo continuo. Así que éste será el mismo, a diferencia de que no deberá estar precedido por un Zero-Holder Hold, ya que la señal recibida está discretizada. Por lo tanto el modelo quedará de la forma que aparece en la Figura 3.7.

3.2.4 DAC.

En este caso será utilizado un bloque *Integer Delay*, que hará exactamente la misma función que en el modelo de tiempo continuo.

3.2.5 Resoluciones.

Como el modelo que está siendo diseñado actualmente es de tiempo discreto, será necesario definir la resolución de cada uno de los elementos que forma el diagrama de bloques para su posterior conversión a un modelo de bloques Xilinx, cuyos valores serán elegidos dependiendo de las operaciones a realizar.

Particularmente, deberán ser elegidas con sumo cuidado las de los elementos del cuantificador (ya que a la salida del modulador deberá obtenerse un código que vaya tomando los valores 7, 5, 3, 1, -1, -3, -5, -7, respectivamente), la de la señal de entrada y la de los términos de corrección de entrada. Con la particularidad de que como la SNR objetivo es de 84 dB, las señales de entrada deberán tener una resolución como mínimo de 14 bits.

Las resoluciones finalemente elegidas son mostradas en el Anexo 2 (Apartado 7.2).

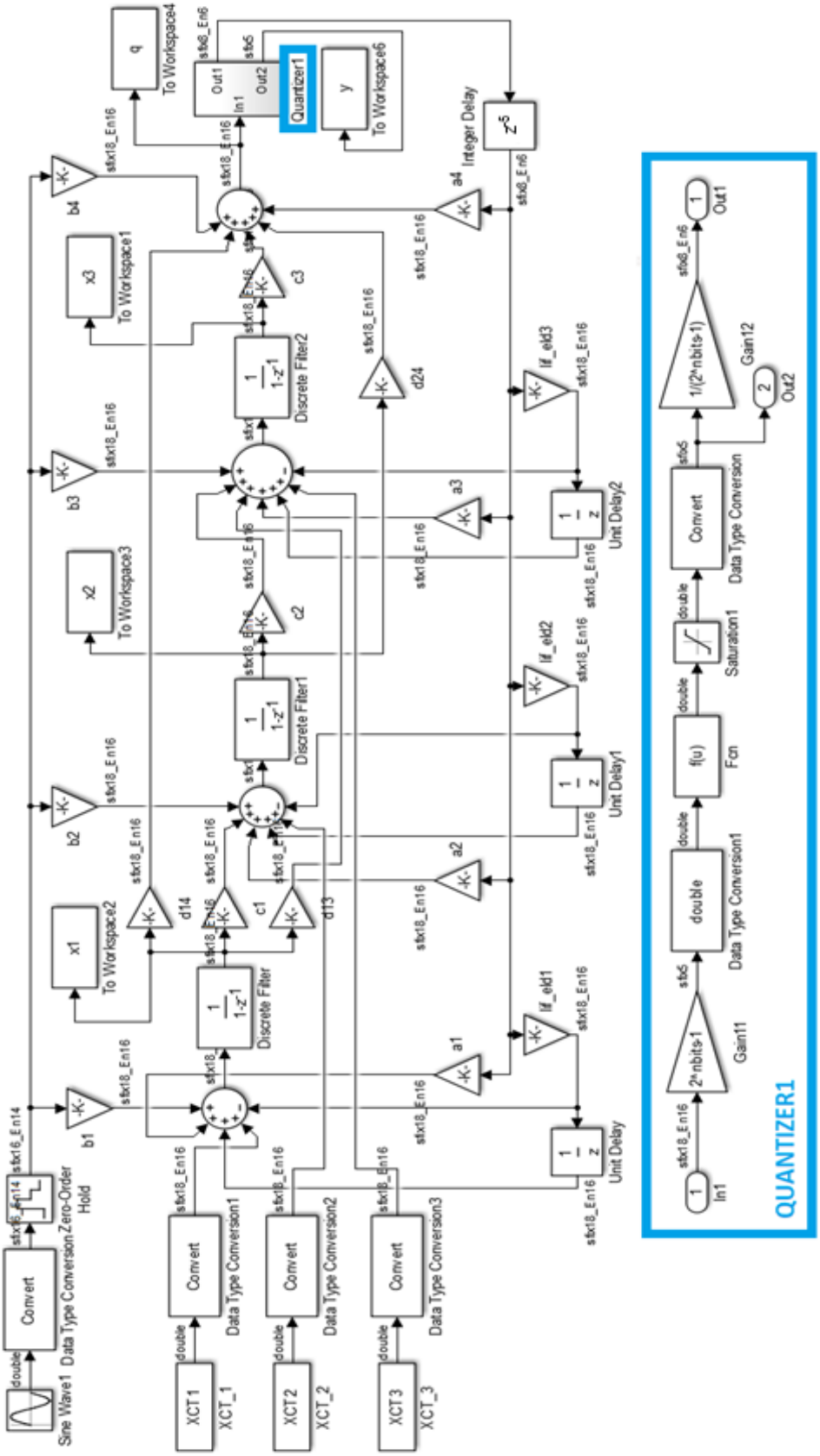

Figura 3.7 – Modelo en tiempo discreto.

3.2.6 Evaluación y conclusión.

Una vez completado el diagrama de bloques del conversor Sigma-Delta de Tiempo Discreto (Figura 3.7), es cuando se deberá simular el modelo para proceder a un exhaustivo análisis en el que se comprobará el correcto funcionamiento del sistema, tal y como se realizó en el diseño del conversor Sigma-Delta de Tiempo Continuo.

Tal y como se realizó en el apartado 3.1.5, en primer lugar se estudiarán las variables de estado representadas en la Figura 3.8, ya que, como se comentó con anterioridad, estas serán el primer indicador de que el sistema funcione o no correctamente.

A continuación, se procede a estudiar la señal de salida ***y***, representada en la Figura 3.9, que en este caso muestra un código decimal que varía de 7 a -7, de dos en dos, periódicamente a lo largo del tiempo. Siendo esto lo que se quería obtener.

Pero, a pesar de todo, las comprobaciones anteriores no serán suficientes para admitir que el diseño es el adecuado. Teniendo que analizar el espectro de frecuencias de la Figura 3.10, al igual que en el diseño del modelo en tiempo continuo, donde se puede observar que la densidad espectral inicial se encuentra un poco distorsionada, debido a la presencia de continua, algo normal es éste caso, y la máxima densidad espectral está aproximadamente a la misma frecuencia que la de la señal de entrada, ***fin***. Por descontado, el ruido volverá a aparecer a partir del límite superior del ancho de banda, representado por la línea roja.

Por último, se estudiará la snr del sistema, siendo ésta de 73.8724 dB. Tal como podemos ver ésta es un poco más pequeña que la obtenida en el modelo de tiempo continuo, pero es algo normal debido a que el error del sistema aumenta al realizar la discretización. A pesar de todo, podemos concluir, que el diseño del Modulador Sigma-Delta de Tiempo Discreto ha sido realizado con éxito y se puede proceder a la última fase de diseño.

NOTA: Es necesario destacar que en este caso se ha considerado una longitud de simulación mayor que en la del modelo anterior, siendo ésta de 2^{15}. Ya que son necesarios más puntos de muestreo para este diseño.

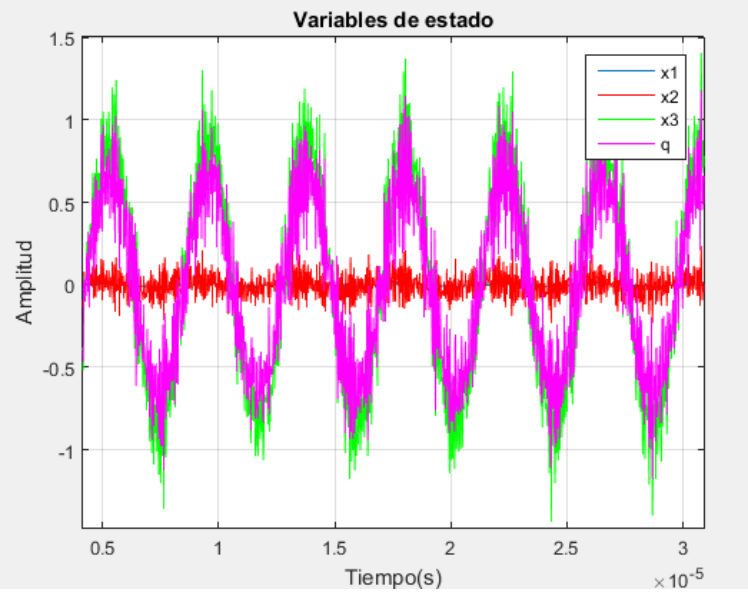

Figura 3.8 - Variables de estado (q, x1, x2 y x3) en función del tiempo (modulador Sigma-Delta de Tiempo Discreto).

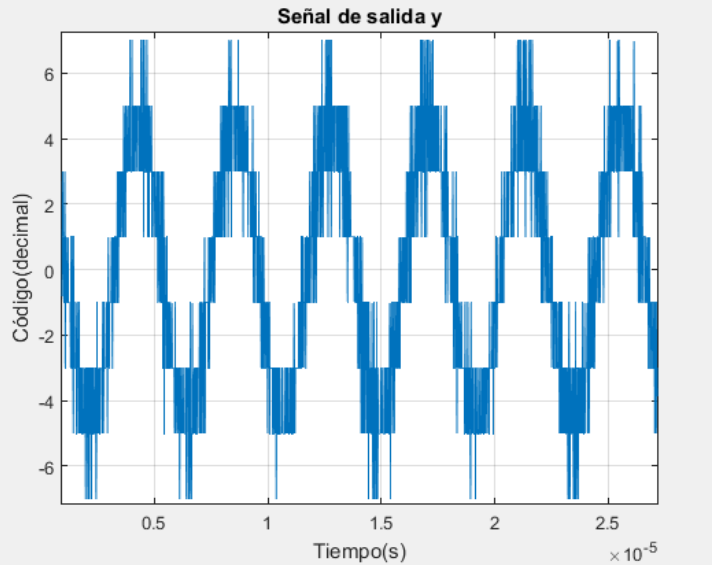

Figura 3.9 - Señal de salida 'y' en función del tiempo (modulador Sigma-Delta de Tiempo Discreto).

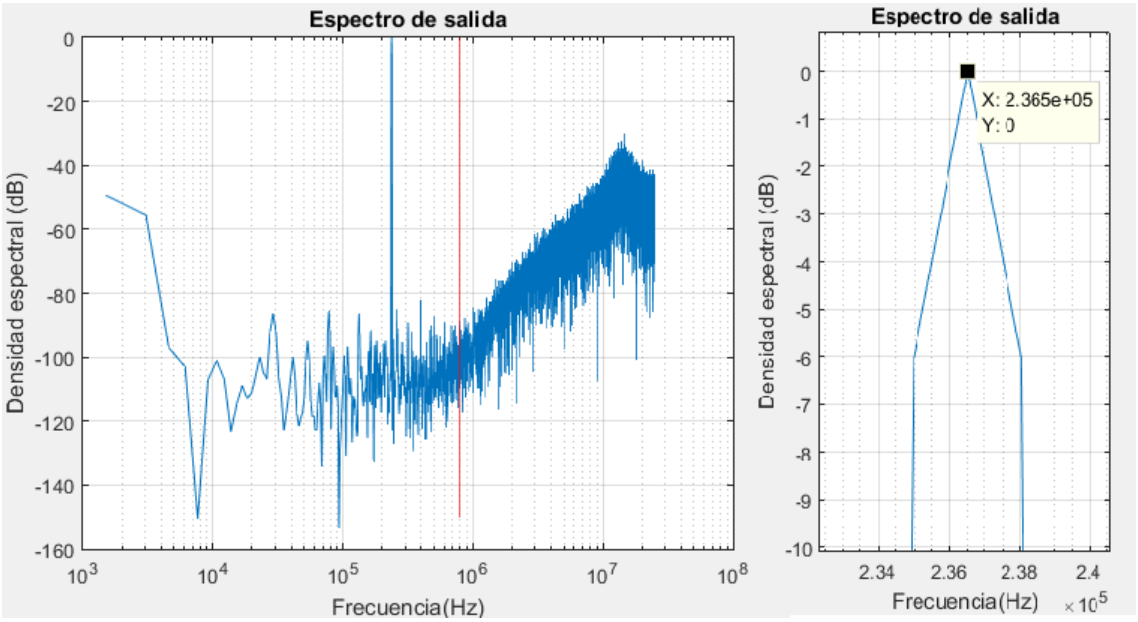

Figura 3.10 - Espectro de salida del modulador (modulador Sigma-Delta de Tiempo Discreto).

3.3 Diseño en bloques de xilinx.

Una vez realizada con satisfacción la discretización del modelo en tiempo continuo, ha llegado el momento de realizar la última fase del diseño, en la que serán realizadas las modificaciones pertinentes para que el modelo pueda ser emulado en ISE.

Cabe destacar que tanto los coeficientes del filtro como los términos de corrección serán exactamente los mismos que los obtenidos para el diseño del modelo de tiempo discreto, Apartados 3.2.1 y 0. Y que los integradores deberán ser transformados tal y como aparece en la Figura 3.11.

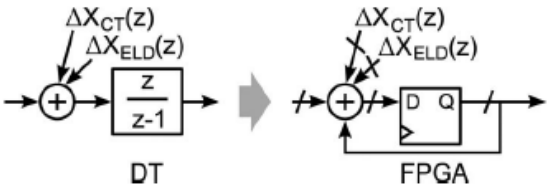

Figura 3.11 – Transformación para emulación en ISE.

3.3.1 Cuantificador y DAC.

En esta ocasión será necesario modificar significativamente el cuantificador, teniendo que diseñar un subsistema como el que aparece en la Figura 3.12, ya que es necesario obtener los valores 7, 5, 3, 1, -1, -3, -5 y -7, a la salida del sistema. Por lo tanto tendrán que ser modificados aquellos códigos que dan como resultado un número par o cero.

Lo único que tendrá que hacerse para deshacer la función del cuantificador es introducir como DAC un Shift de sentido contrario al del cuantificador. Obteniendose un sistema final tal y como se muestra en la Figura 3.13.

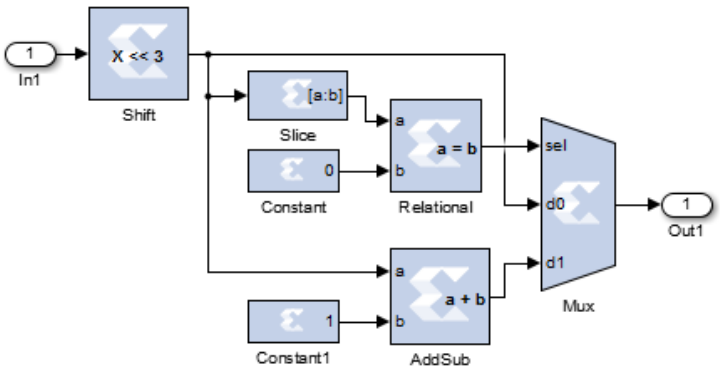

Figura 3.12– Cuantificador modelo Bloques Xilinx.

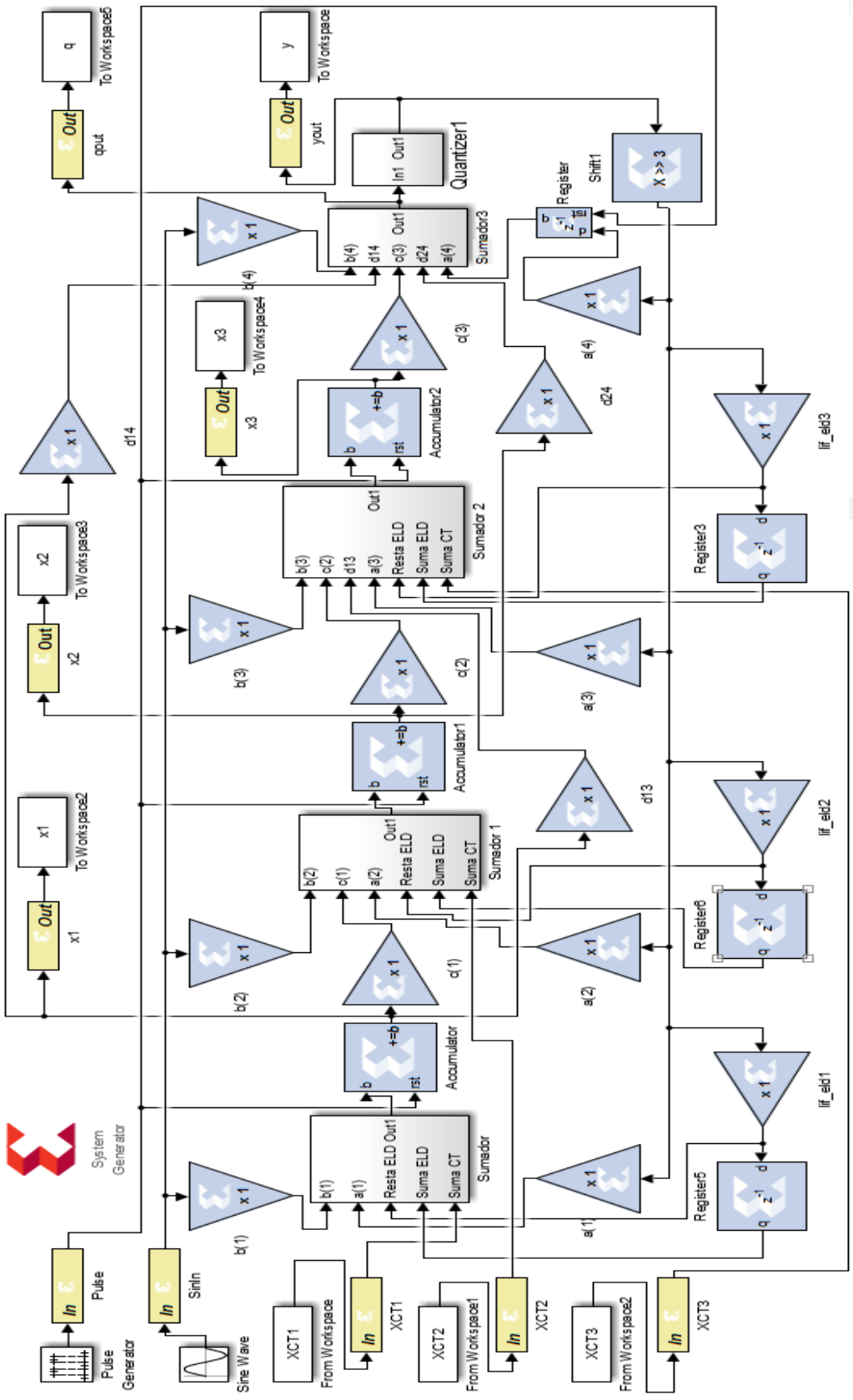

Figura 3.13 – Modelo Bloques de Xilinx.

4 EVALUACIÓN DEL DISEÑO FINAL.

A diferencia de los diseños de tiempo continuo y tiempo discreto, en este tuvo que ser realizado un análisis más exhaustivo del diseño en Bloques Xilinx, ya que se trata del diseño final del proyecto y es muy importante que funcione correctamente.

4.1 Evaluación mediante MATLAB/Simulink.

En primer lugar se realizará una emulación en MATLAB/Simulink, ya que si el sistema no se comporta correctamente en él, el diseño será erróneo y sería necesario modificarlo hasta lograr su correcto funcionamiento.

Tal y como se realizó en el apartado 0, en primer lugar tendrán que ser estudiadas las variables de estado que han sido elegidas, las cuales aparecen representadas en la Figura 4.1.

A continuación, se procederá a estudiar la señal de salida ***y***, representada en la Figura 3.9, la cual muestra un código decimal que varía de 7 a -7, de dos en dos, periódicamente a lo largo del tiempo. Siendo esto lo que se quería obtener.

Es necesario también analizar el espectro de frecuencias de la Figura 4.3, al igual que en los diseños anteriores, donde se puede observar que la densidad espectral inicial se encuentra nuevamente distorsionada, debido a la presencia de continua, y la máxima densidad espectral vuelve a estar aproximadamente a la misma frecuencia que la de la señal de entrada, ***fin***. Apareciendo nuevamente el ruido a partir del límite superior del ancho de banda, representado por la línea roja.

Por último, se estudiará la SNR del sistema, siendo ésta de 74.6486 dB, que sigue siendo algo más pequeña que la del modelo en tiempo continuo, pero aún así esta es un poco mejor que la del modelo en tiempo discreto. A pesar de ello se puede concluir, que el diseño del Modulador en Bloques Xilinx ha sido aparentemente realizado con éxito, algo que terminará de comprobarse en el siguiente apartado.

NOTA: Únicamente recordar que la longitud de simulación tomada ha sido exactamente la misma que la del diseño del modelo en tiempo discreto, siendo ésta de 2^{15}.

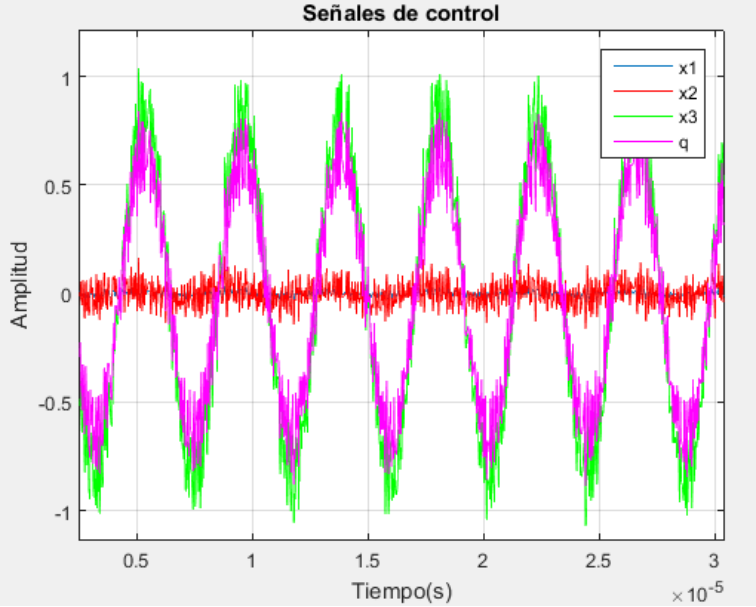

Figura 4.1 – Variables de estado (q, x1, x2 y x3) en función del tiempo (modelo en Bloques Xilinx).

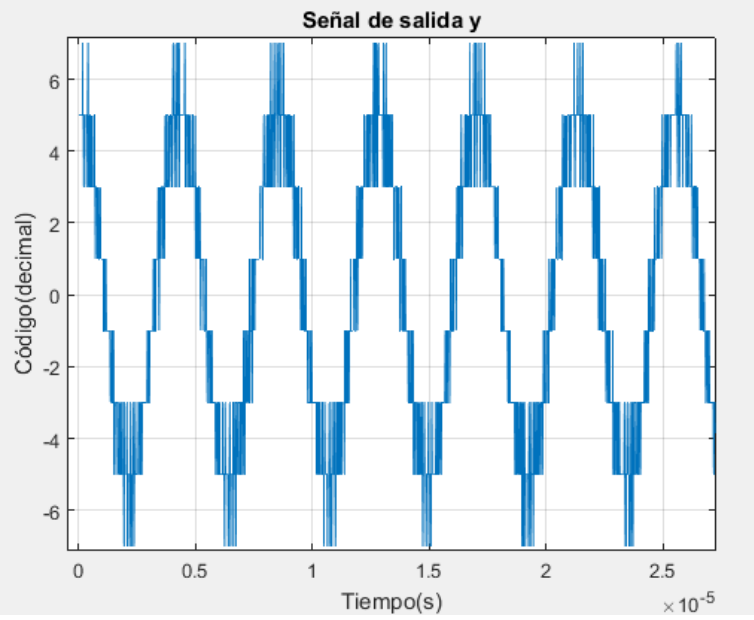

Figura 4.2 – Señal de salida 'y' en función del tiempo (modelo en Bloques Xilinx).

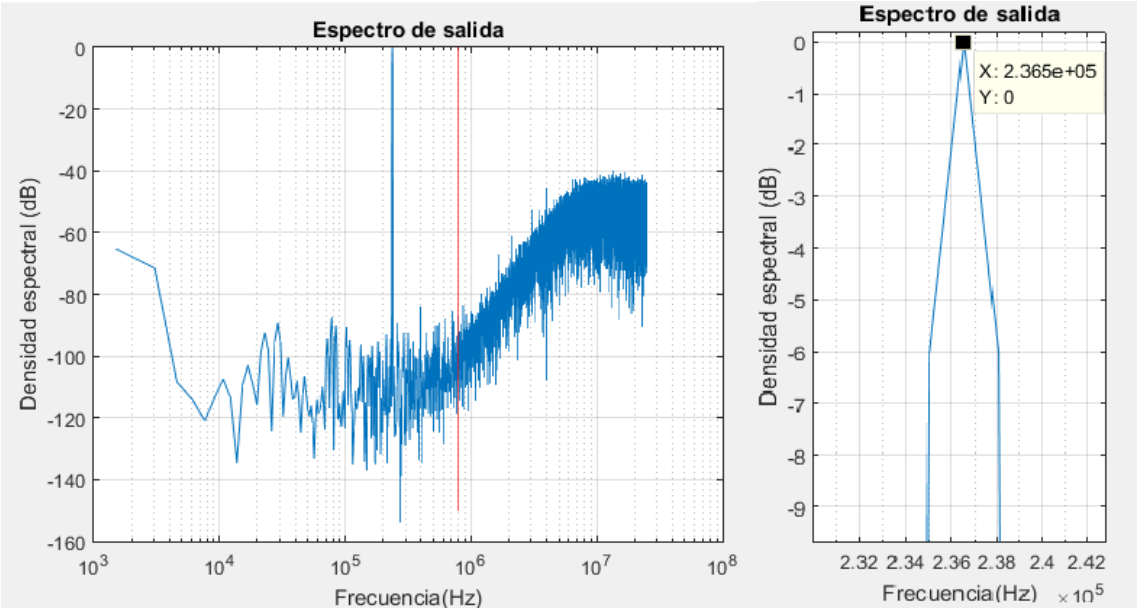

Figura 4.3 – Espectro de salida del modulador (modelo en Bloques Xilinx).

4.2 Evaluación mediante ISE.

Una vez comprobado que el comportamiento del modelo es correcto en MATLAB/Simulink, ha llegado el momento de realizar el siguiente paso, que será generar el HDL Code del sistema mediante System Generator, para poder realizar la emulación a través de ISE. Lo que realizará este Software será una comprobación de que los datos obtenidos son los mismos que a través de MATLAB/Simulink, y generará un error en el caso de que esto no sea así.

Por suerte, al realizar la simulación, se obtuvo un resultado satisfactorio, tal y como puede verse en la captura de la Figura 4.4. Donde se ha realizado un seguimiento de las señales internas y externas del sistema en valor decimal. Tal como puede verse, la salida ***y*** comienza varía, tomando los valores experados. Pudiendo concluir finalmente que el sistema ha sido diseñado con éxito.

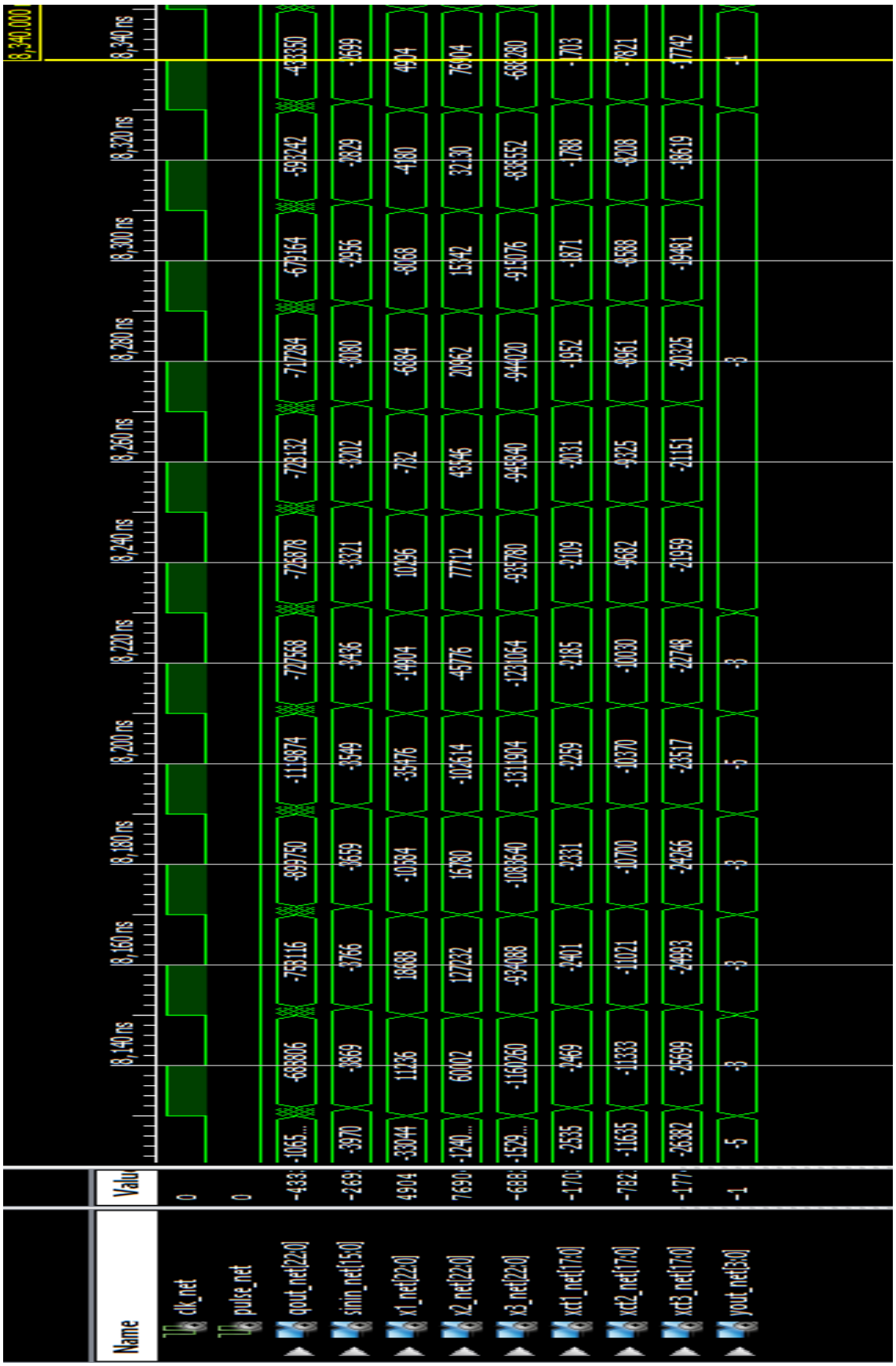

Figura 4.4 – Captura de pantalla ISE.

5 PRESUPUESTO.

A continuación se recogen los costes globales de la realización de éste proyecto fin de carrera (Tabla 3).

FASES	HORAS
Documentación	200
Desarrollo del sistema propuesto y pruebas de funcionamiento	500
Redacción de la memoria del proyecto	200

Tabla 3 - Fases del proyecto

Teniendo en cuenta que el Colegio Oficial de Ingenieros Industriales establece una tarifa de 24€/hora, el coste ascendería a 21600 €.

El resto de gastos se recogen en la Tabla 4.

Concepto	Importe(€)
Ordenador portátil	1000

Tabla 4 – Costes de material.

A partir de los datos obtenidos, el presupuesto total asciende a 22600 € sin IVA y 26668 € con IVA.

NOTA: No se han incluido los costes de los softwares utilizados para la realización de este proyecto, ya que tanto MATLAB como ISE han sido utilizados mediante una licencia gratuita proporcionada por la Universidad Carlos III de Madrid.

6 CONCLUSIONES Y TRABAJOS FUTUROS.

El principal objetivo de este proyecto consistía en implementar un modelo hardware de simulación para modulares Sigma-Delta de Tiempo Continuo que permitiera acelerar el proceso de simulación de estos circuitos utilizando una FPGA, pero esto no ha podido realizarse por completo ya que el sistema no ha podido ser implementado en la FPGA. Esto ha sido debido a la complejidad que ha supuesto el diseño de los modelos, que ha llevado demasiado tiempo, y a la complejidad de realizar la implementación en una FPGA.

A pesar de todo, puedo decir el trabajo realizado ha sido satisfactorio, ya que el modelo diseñado es este proyecto cumple con los requisitos y gracias a las comparaciones realizadas con el estudio [1], se puede decir que los resultados han sido bastante parecidos. Estos no han podido ser iguales, ya que en el caso de estudio diseñaron un modulador de tercer orden con un cuantificador de 4bits y 50% de ELD, mientras que el modelo diseñado en este proyecto ha sido un modulador de tercer orden con 50% de ELD, pero con un cuantificador de 3bits. Aún así, al comparar la resolución espectral obtenida en nuestro modelo (Figura 4.3) con la que obtuvieron en el artículo de estudio (Figura 2.6), puede verse que son bastante similares: apareciendo la máxima densidad espectral a la frecuencia de la señal de entrada y comenzando el ruido térmico a partir del límite superior del ancho de banda. Lo único que haría falta, como he mencionado antes, sería realizar la implementación del modelo en una FPGA y comprobar que realmente el tiempo de simulación disminuye en comparación con las simulaciones realizadas en MATLAB/SIMULINK.

Por lo tanto, en trabajos futuros pueden incluirse las siguientes líneas de estudio:

- Implementación del sistema en una FPGA, para que pueda comunicarse con un ordenador.
- Comprobación de que la emulación anterior resulta ser más rápida que la simulación que se realiza a través de MATLAB/Simulink.
- Modificación del diseño para que puedan utilizarse otros tipos de señales como señal de entrada, y que esta no sea únicamente senosoidal.

En conclusión, el trabajo realizado en este proyecto ha resultado satisfactorio y me ha permitido ampliar mis conocimientos sobre MATLAB/Simulink, ISE Desing Suite y Sistemas de Control. Siendo este último un campo de aplicación amplio y con grandes perpectivas de futuro.

7 ANEXOS.

7.1 ANEXO 1 – DISEÑO TIEMPO CONTINUO.

tiempo_continuo.m

```
clear all

OSR=32;             % relacion sobremuestreo OSR=fs/(2BW)
hinf=2;
order=3;            % orden del convertidor (esto quiere decir que el
                    % convertidor tendrá tres integradores).
nbits=3;            % número de bits (relacionado con el orden del
                    % modulador).
fs=50e6;            % frecuencia de muestreo.
Ts=1/fs;            % período de muestreo.
BW=fs/2/OSR;        % ancho de banda.
lsim=2^13;          % longitud de simulación (es un valor expresado
                    % como 2^x, donde x ha sido elegida como x=13).
i=1;                % variable de conteo para obtener a, b, c y d.

% Características de la señal de entrada al circuito.
ampin = 10^(-10/20); % Ponemos la amplitud pero en Hercios.
binfin=round((0.3*BW)/fs*lsim);
fin=binfin/lsim*fs;
win=2*pi*fin;

ntf=synthesizeNTF(order,OSR,0,hinf,0);
% con ella se obtiene la función de transferencia, a pesar de no tener
% el modulador diseñado aún, porque es necesaria para la obtención de
% la matriz ABCD mediante:

[ABCDc]   =   realizeNTF_ct(ntf,'FB',{[0.5,1.5];[0.5,1.5];   [0.5,1.5];
[0.5,1.5]});
% de esta matriz se obtiene el valor de los coeficientes necesarios:

while i<=4,
    if i~= 4
    c(i)= ABCDc(i+1,i);
    end
    if i == 1
        a(i)= ABCDc(i,i+4);
        b(i)= ABCDc(i,i+3);
    else
        a(i)= ABCDc(i,i+4);
        b(i)= -a(i);
    end
```

```
    i=i+1;
end

%Simulación del circuito:
sim('tiempo_continuoc', 50*Ts+lsim*Ts)


% Representación de señales de control y espectro de salida para
% comprobar el correcto funcionamiento del sistema:
y=y(end-lsim+1:end);
yf=fft(y'.*ds_hann(lsim));
yf=yf(1:lsim/2);
binfft=1:lsim/2;
freq=binfft*(fs/lsim);

figure(1),clf
semilogx(freq,20*log10(abs(yf)/max(abs(yf)))), grid on, hold on
semilogx([BW BW], [-150 0],'r')
xlabel('Frecuencia(Hz)')
ylabel('Densidad espectral (dB)')
title('Espectro de salida')

figure(2), clf
los=1:lsim;
tiempo=los*Ts;
plot(tiempo,y), grid on
xlabel('Tiempo(s)')
ylabel('Amplitud')
title('Señal de salida y')

figure(3), clf
x1=x1(1:lsim);
x2=x2(1:lsim);
x3=x3(1:lsim);
q=q(1:lsim);
plot(tiempo,x1), grid on
hold on
plot(tiempo,x2,'r')
plot(tiempo,x3,'g')
plot(tiempo,q,'m')
legend('x1','x2','x3','q')
xlabel('Tiempo(s)')
ylabel('Amplitud')
title('Variables de estado')

% Cálculo de snr pra comprobar el correcto funcionamiento del sistema:
sig=yf(binfin:binfin+2);
noi=yf(1:lsim/2/OSR);
noi(binfin:binfin+2)=[0 0 0];
snr=10*log10((sum(abs(sig).^2))/(sum(abs(noi).^2)))
```

7.2 ANEXO 2 – DISEÑO TIEMPO DISCRETO.

tiempo_discreto.m

```
clear all

OSR=32;             % Relacion sobremuestreo OSR=fs/(2BW)
hinf=2;
order=3;            % Orden del convertidor (esto quiere decir que el
                    % convertidor tendrá tres integradores).
nbits=3;            % Número de bits (relacionado con el orden del
                    % modulador).
fs=50e6;            % Frecuencia de muestreo.
Ts=1/fs;            % Período de muestreo.
BW=fs/2/OSR;        % Ancho de banda.
lsim=2^15;          % longitud de simulación (es un valor expresado como
                    % 2^x,donde x ha sido elegida como x=15, en lugar de
                    % 13, para tener más puntos de muestreo).
tsim = lsim*Ts;     % tiempo de simulación.
k=1;                % variable de conteo para obtener las matrices.

% Características de la señal de entrada al circuito.
ampin = 10^(-10/20); % Amplitud, pero en Hercios.
binfin=round((0.3*BW)/fs*lsim);
fin=binfin*(fs/lsim);
win=2*pi*fin;

%%RESOLUCIONES%%-----------------------------------------------------

 % SEÑALES DE ENTRADA:
 in_bits = 16;
 in_dec = 14;

 XCT_bits = 18;
 XCT_dec = 16;

 % FILTROS (LOS QUE ESTAN TRAS LOS SUMADORES):
 filter_bits = 18;
 filter_dec = 16;

 % GANANCIAS A, B, C, D:
 b_bits = 8;     % Resolución de operación interna.
 b_dec = 6;

 bo_bits = 18;   % Resolución de salida.
 bo_dec = 16;

 a_bits = 8;
```

```
a_dec = 6;

ao_bits = 18;
ao_dec = 16;

c_bits = 2;
c_dec = 0;

co_bits = 18;
co_dec = 16;

d_bits = 3;
d_dec = 1;

do_bits = 18;
do_dec = 16;

eld_bits = 8;
eld_dec = 6;

eld1o_bits = 18;
eld1o_dec = 16;

% SUMADORES:
sum_bits = 18;
sum_dec = 16;

% QUANTIZER:
gain11_bits = 5;
gain11_dec = 0;

gain11o_bits = 5;
gain11o_dec = 0;

gain12_bits = 8;
gain12_dec = 6;

gain12o_bits = 8;
gain12o_dec = 6;

%%_____________________________________________________________________

ntf=synthesizeNTF(order,OSR,0,hinf,0);

% Cálculo de las MATRICES de TIEMPO CONTINUO, ya que van a ser
% necesarias para la obtención de las de TIEMPO DISCRETO:

[ABCDc] = realizeNTF_ct(ntf,'FB',{[0.5,1.5];[0.5,1.5]; [0.5,1.5];
[0.5,1.5]});
```

```
while k<=4,
    if k == 1
        ac(k)= ABCDc(k,k+4);
        bc(k)= ABCDc(k,k+3);
    else
        ac(k)= ABCDc(k,k+4);
        bc(k)= -ac(k);
    end
    k=k+1;
end

A = ABCDc(1:3,1:3);
B = [bc(1:3); ac(1:3)]';
C = ABCDc(4,1:3);
D = [bc(4); ac(4)]';

% TRANSFORMACIÓN DE ORDEN CERO (ZOH), con las fórmulas que se indican
% en el artículo:

Adt=expm(A);
Cdt=C;
Ddt=D;

syms x;
f = expm(A*(1-x))*B;
Bdt = int(f,0,1);

% Con ellas se obtienen los nuevos coeficientes a, b, c y d
% NOTA: El double es necesario porque en un principio los valores no
% se obtienen en ese formato y no pueden ser utilizados en el
% circuito. De este modo se soluciona el problema.

a = [double(Bdt(1,2)) double(Bdt(2,2)) double(Bdt(3,2)) Ddt(2)];
b = [double(Bdt(1,1)) double(Bdt(2,1)) double(Bdt(3,1)) Ddt(1)];
c = [Adt(2,1) Adt(3,2) Cdt(3)];
d = [Adt(3,1) Cdt(1) Cdt(2)];

%lifting terms
%XELD
syms x;
f = expm(A*(1-x))*B(:,2);    %B de las salidas (r(n))
Xeld = int(f,0,0.5);
lif_eld=double(Xeld);

%XCT
syms w;
g = expm(A*(1-w))*B(:,1)*exp(w*win*Ts*1i);
```

```
h = int(g,w,0,1);
imagina = double(h);

n=(0:lsim+55)';
XCT1.signals.values = ampin*sin(n*win*Ts)*real(imagina(1))+
                      ampin*cos(n*win*Ts)*imag(imagina(1));
XCT2.signals.values = ampin*sin(n*win*Ts)*real(imagina(2))+
                      ampin*cos(n*win*Ts)*imag(imagina(2));
XCT3.signals.values = ampin*sin(n*win*Ts)*real(imagina(3))+
                      ampin*cos(n*win*Ts)*imag(imagina(3));

XCT1.time=n*Ts;
XCT2.time=n*Ts;
XCT3.time=n*Ts;
XCT1.signals.dimensions=1;
XCT2.signals.dimensions=1;
XCT3.signals.dimensions=1;

% Simulación del circuito:
sim('tiempo_discretoc', 50*Ts+lsim*Ts)

% Representación de señales de control y espectro de salida para
% comprobar el correcto funcionamiento del sistema:
y=y(end-lsim+1:end);
yf=fft(y'.*ds_hann(lsim));
yf=yf(1:lsim/2);
binfft=1:lsim/2;
freq=binfft*(fs/lsim);

figure(4),clf
semilogx(freq,20*log10(abs(yf)/max(abs(yf)))), grid on, hold on
semilogx([BW BW], [-150 0],'r')
xlabel('Frecuencia(Hz)')
ylabel('Densidad espectral (dB)')
title('Espectro de salida')

figure(5), clf
los=1:lsim;
tiempo=los*Ts;
plot(tiempo,y), grid on
xlabel('Tiempo(s)')
ylabel('Código(decimal)')
title('Señal de salida y')

figure(6), clf
x1=x1(1:lsim);
x2=x2(1:lsim);
x3=x3(1:lsim);
q=q(1:lsim);
plot(tiempo,x1), grid on
hold on
```

```
plot(tiempo,x2,'r')
plot(tiempo,x3,'g')
plot(tiempo,q,'m')
legend('x1','x2','x3','q')
xlabel('Tiempo(s)')
ylabel('Amplitud')
title('Variables de estado')

% Cálculo de snr pra comprobar el correcto funcionamiento del sistema:
sig=yf(binfin:binfin+2);
noi=yf(1:lsim/2/OSR);
noi(binfin:binfin+2)=[0 0 0];
noi(1:2)=[0 0];
snr=10*log10((sum(abs(sig).^2))/(sum(abs(noi).^2)))
```

7.3 ANEXO 3 – DISEÑO BLOQUES XILINX.

tiempo_FPGA.m

```
clear all

OSR=32;             % Relacion sobremuestreo OSR=fs/(2BW)
hinf=2;
order=3;            % Orden del convertidor (esto quiere decir que en el
                    % convertidor
                    % vamos a tener tres integradores).
nbits=3;            % Número de bits (relacionado con el orden del
                    % modulador).
fs=50e6;            % Frecuencia de muestreo.
Ts=1/fs;            % Período de muestreo.
BW=fs/2/OSR;        % Ancho de banda.
lsim=2^15;          % longitud de simulación (es un valor expresado como
                    % 2^x, donde x ha sido elegida como x=15, en lugar de
                    % 13,para tener más puntos de muestreo).
tsim = lsim*Ts;     % Tiempo de simulación que vamos a utilizar.
k=1;                % Variable de conteo que utilizaremos para la
                    % obtención de las matrices que necesitamos.

% Características de la señal de entrada al circuito.
ampin = 10^(-10/20); % Amplitud, pero en Hercios.
binfin=round((0.3*BW)/fs*lsim);
fin=binfin*(fs/lsim);
win=2*pi*fin;

ntf=synthesizeNTF(order,OSR,0,hinf,0);

% Cálculo de las MATRICES de TIEMPO CONTINUO, ya que van a ser
% necesarias para la obtención de las de TIEMPO DISCRETO:
[ABCDc] = realizeNTF_ct(ntf,'FB',{[0.5,1.5];[0.5,1.5]; [0.5,1.5];
[0.5,1.5]});

while k<=4,
    if k == 1
        ac(k)= ABCDc(k,k+4);
        bc(k)= ABCDc(k,k+3);
    else
        ac(k)= ABCDc(k,k+4);
        bc(k)= -ac(k);
    end
    k=k+1;
end

A = ABCDc(1:3,1:3);
```

```
B = [bc(1:3); ac(1:3)]';
C = ABCDc(4,1:3);
D = [bc(4); ac(4)]';
% TRANSFORMACIÓN DE ORDEN CERO (ZOH), con las fórmulas que se indican
% en el artículo:

Adt=expm(A);
Cdt=C;
Ddt=D;

syms x;
f = expm(A*(1-x))*B;
Bdt = int(f,0,1);

% Con ellas se obtienen los nuevos coeficientes a, b, c y d
% NOTA: El double es necesario porque en un principio los valores no
% se obtienen en ese formato y no pueden ser utilizados en el
% circuito. De este modo se soluciona el problema.

a = [double(Bdt(1,2)) double(Bdt(2,2)) double(Bdt(3,2)) Ddt(2)];
b = [double(Bdt(1,1)) double(Bdt(2,1)) double(Bdt(3,1)) Ddt(1)];
c = [Adt(2,1) Adt(3,2) Cdt(3)];
d = [Adt(3,1) Cdt(1) Cdt(2)];

% lifting terms
% XELD
syms x;
f = expm(A*(1-x))*B(:,2);    %B de las salidas (r(n))
Xeld = int(f,0,0.5);
lif_eld=double(Xeld);

%XCT
syms w;
g = expm(A*(1-w))*B(:,1)*exp(w*win*Ts*1i);
h = int(g,w,0,1);
imagina = double(h);

n=(0:lsim+55)';
XCT1.signals.values = ampin*sin(n*win*Ts)*real(imagina(1))+
                      ampin*cos(n*win*Ts)*imag(imagina(1));
XCT2.signals.values = ampin*sin(n*win*Ts)*real(imagina(2))+
                      ampin*cos(n*win*Ts)*imag(imagina(2));
XCT3.signals.values = ampin*sin(n*win*Ts)*real(imagina(3))+
                      ampin*cos(n*win*Ts)*imag(imagina(3));

XCT1.time=n*Ts;
XCT2.time=n*Ts;
XCT3.time=n*Ts;
XCT1.signals.dimensions=1;
XCT2.signals.dimensions=1;
XCT3.signals.dimensions=1;
```

```
%Simulamos el circuito.
sim('tiempo_discretoc', 50*Ts+lsim*Ts)
% Representación de las señales a estudiar para comprobar el correcto
% funcionamiento del sistema.
y=y(end-lsim+1:end);
yf=fft(y'.*ds_hann(lsim));
yf=yf(1:lsim/2);
binfft=1:lsim/2;
freq=binfft*(fs/lsim);

figure(7),clf
semilogx(freq,20*log10(abs(yf)/max(abs(yf)))), grid on, hold on
semilogx([BW BW], [-150 0],'r')
xlabel('Frecuencia(Hz)')
ylabel('Densidad espectral (dB)')
title('Espectro de salida')

figure(8), clf
los=1:lsim;
tiempo=los*Ts;
plot(tiempo,y), grid on
xlabel('Tiempo(s)')
ylabel('Código(decimal)')
title('Señal de salida y')

figure(9), clf
x1=x1(1:lsim);
x2=x2(1:lsim);
x3=x3(1:lsim);
q=q(1:lsim);
plot(tiempo,x1), grid on
hold on
plot(tiempo,x2,'r')
plot(tiempo,x3,'g')
plot(tiempo,q,'m')
legend('x1','x2','x3','q')
xlabel('Tiempo(s)')
ylabel('Amplitud')
title('Variables de estado')

% Cálculo de la snr para comprobar el correcto funcionamiento del
% sistema.
sig=yf(binfin:binfin+2);
noi=yf(1:lsim/2/OSR);
noi(binfin:binfin+2)=[0 0 0];
noi(1:2)=[0 0];
snr=10*log10((sum(abs(sig).^2))/(sum(abs(noi).^2)))
```

7.4 ANEXO 4 – FUNCIONES NECESARIAS.

calculateSNR.m

```
function snr = calculateSNR(hwfft,f,nsig)
% snr = calculateSNR(hwfft,f,nsig=1) Estimate the signal-to-noise
ratio,
% given the in-band bins of a (Hann-windowed) fft and
% the location of the input signal (f>0).
% For nsig=1, the input tone is contained in hwfft(f:f+2);
%  this range is appropriate for a Hann-windowed fft.
% Each increment in nsig adds a bin to either side.
% The SNR is expressed in dB.
if nargin<3
    nsig = 1;
end
signalBins = [f-nsig+1:f+nsig+1];
signalBins = signalBins(signalBins>0);
signalBins = signalBins(signalBins<=length(hwfft));
s = norm(hwfft(signalBins));        % *4/(N*sqrt(3)) for true rms
value;
noiseBins = 1:length(hwfft);
noiseBins(signalBins) = [];
n = norm(hwfft(noiseBins));
if n==0
    snr = Inf;
else
    snr = dbv(s/n);
end
```

dbv.m

```
function y=dbv(x)
% dbv(x) = 20*log10(abs(x)); the dB equivalent of the voltage x
y = -Inf*ones(size(x));
if isempty(x)
    return
end
nonzero = x~=0;
y(nonzero) = 20*log10(abs(x(nonzero)));
```

ds_hann.m

```
function w = ds_hann(n)

% function w = ds_hann(n)
% A Hann window of length n. Does not smear tones located exactly in a
bin.
% Note: This function was formerly just "hann." Re-naming
% was necessary to avoid a conflict with Mathworks's function
% of the same name.
w = .5*(1 - cos(2*pi*(0:n-1)/n) );
```

evalTF.m

```
function h = evalTF(tf,z)
%h = evalTF(tf,z)
%Evaluates the rational function described by the struct tf
% at the point(s) given in the z vector.
% TF must be either a zpk object or a struct containing
%   form        'zp' or 'coeff'
%   zeros,poles,k   if form=='zp'
%   num,den     if form=='coeff'
%
% In Matlab 5, the ss/freqresp() function does nearly the same thing.

if isobject(tf)     % zpk object
    if strcmp(class(tf),'zpk')
    h = tf.k * evalRPoly(tf.z{1},z) ./ evalRPoly(tf.p{1},z);
    else
    fprintf(1,'%s: Only zpk objects supported.\n', mfilename);
    end
elseif any(strcmp(fieldnames(tf),'form'))
    if strcmp(tf.form,'zp')
    h = tf.k * evalRPoly(tf.zeros,z) ./ evalRPoly(tf.poles,z);
    elseif strcmp(tf.form,'coeff')
    h = polyval(tf.num,z) ./ polyval(tf.den,z);
    else
    fprintf(1,'%s: Unknown form: %s\n', mfilename, tf.form);
    end
else    % Assume zp form
    h = tf.k * evalRPoly(tf.zeros,z) ./ evalRPoly(tf.poles,z);
end
```

evalRPoly.m

```
function y = evalRPoly(roots,x,k)
%function y = evalRPoly(roots,x,k=1)
%Compute the value of a polynomial which is given in terms of its
roots.
if(nargin<3)
    k=1;
end
y = k(ones(size(x)));
roots = roots(~isinf(roots));        % remove roots at infinity
for(i=1:length(roots))
    y = y.*(x-roots(i));
end
```

evalTFP.m

```
function H=evalTFP(Hs,Hz,f)
% H=evalTFP(Hs,Hz,f)
% Compute the value of a transfer function product Hs*Hz at a
frequency f,
% where Hs is a cts-time TF and Hz is a discrete-time TF.
% Both Hs and Hz are SISO zpk objects.
% This function attempts to cancel poles in Hs with zeros in Hz.

szeros = Hs.z{1};
spoles = Hs.p{1};
zzeros = Hz.z{1};
zpoles = Hz.p{1};

slim = min(1e-3,max(1e-5,eps^(1/(1+length(spoles)))));
zlim = min(1e-3,max(1e-5,eps^(1/(1+length(zzeros)))));

H = zeros(size(f));
w = 2*pi*f; s = j*w;    z=exp(s);
for i=1:length(f)
    wi = w(i);  si = s(i);  zi = z(i);
    if isempty(spoles)
        cancel = 0;
    else
        cancel = abs(si-spoles)<slim;
    end
    if ~cancel
        % wi is far from a pole, so just use the product Hs*Hz
        H(i) = evalTF(Hs,si) * evalTF(Hz,zi);
    else
        % cancel pole(s) of Hs with corresponding zero(s) of Hz
        cancelz = abs(zi-zzeros)<zlim;
        if sum(cancelz) > sum(cancel)
            H(i) = 0;
        elseif sum(cancelz) < sum(cancel)
                H(i) = Inf;
        else
            H(i) =  evalRPoly(szeros,si,Hs.k) * ...
                zi^sum(cancel) * evalRPoly(zzeros(~cancelz),zi,Hz.k) /
            (evalRPoly(spoles(~cancel),si,1)*evalRPoly(zpoles,zi,1));
        end
    end
end
```

impL1.m

```
function y = impL1(arg1,n)
% y=impL1(ntf,n=10)
% Compute the impulse response from the comparator
% output to the comparator input for the given NTF.
% n is the (optional) number of points (10).
%
% This function is useful when verifying the realization
% of a NTF with a specified topology.
if nargin<2
    n=10;
end
if isobject(arg1) & strcmp(class(arg1),'zpk')
    z = arg1.z{1};
    p = arg1.p{1};
elseif isstruct(arg1)
    if any(strcmp(fieldnames(arg1),'zeros'))
    z = arg1.zeros;
    else
    error('No zeros field in the NTF.')
    end
    if any(strcmp(fieldnames(arg1),'poles'))
    p = arg1.poles;
    else
    error('No poles field in the NTF.')
    end
end

lf_den = padr(poly(z),length(p)+1);
lf_num = lf_den-poly(p);
if any(imag([lf_num lf_den]))
    % Complex loop filter
    lfr_den = real( conv(lf_den,conj(lf_den)) );
    lfr_num = conv(lf_num,conj(lf_den));
    lf_i = tf( real(lfr_num), lfr_den, 1);
    lf_q = tf( imag(lfr_num), lfr_den, 1);
    y = impulse(lf_i,n) + 1i * impulse(lf_q,n);
else
    y = impulse(tf(lf_num,lf_den,1),n);
end
```

padb.m

```
function y = padb(x, n, val)
%y = padb(x, n, val)
% Pad a matrix x on the bottom to length n with value val(0)
% The empty matrix is assumed to be have 1 empty column
if nargin < 3
    val = 0;
end

y = [ x ; val( ones( n-size(x,1), max(1,size(x,2)) ) ) ];
```

padr.m

```
function y = padr(x, n, val)
%y = padr(x, n, val)
% Pad a matrix x on the right to length n with value val(0)
% The empty matrix is assumed to be have 1 empty row
if nargin < 3
    val = 0;
end

y = [ x val(ones(max(1,size(x,1)),n-size(x,2))) ];
```

pulse.m

```
function y = pulse(S,tp,dt,tfinal,nosum)
% y = pulse(S,tp=[0 1],dt=1,tfinal=10,nosum=0)  Calculate the sampled
pulse response
% of a ct system. tp may be an array of pulse timings, one for each
input.
% Outputs
% y The pulse response
%
% Inputs
% S An LTI object specifying the system.
% tp    An nx2 array of pulse timings
% dt    The time increment
% tfinal    The time of the last desired sample
% nosum A flag indicating that the responses are not to be summed

% Handle the input arguments
parameters = {'S','tp','dt','tfinal','nosum'};
defaults = { NaN, [0 1], 1, 10, 0};
for i=1:length(defaults)
    parameter = char(parameters(i));
    if i>nargin | ( eval(['isnumeric(' parameter ') ']) &  ...
     eval(['any(isnan(' parameter ')) | isempty(' parameter ') ']) )
```

```
    eval([parameter '=defaults{i};'])
    end
end
% Check the arguments
if S.Ts ~= 0
    fprintf(1, 'Error: S must be a cts-time system.\n');
    return
end

% Compute the time increment
dd = 1;
for i=1:prod(size(tp))
    [x di] = rat(tp(i),1e-3);
    dd = lcm(di,dd);
end
[x ddt] = rat(dt,1e-3);
[x df] = rat(tfinal,1e-3);
delta_t = 1 / lcm( dd, lcm(ddt,df) );
delta_t = max(1e-3, delta_t);   % Put a lower limit on delta_t
y1 = step(S,0:delta_t:tfinal);

nd = round(dt/delta_t);
nf = round(tfinal/delta_t);
ndac = size(tp,1);
ni = size(S.b,2);
if rem(ni,ndac)~=0
    error('The number of inputs must be divisible by the number of dac
timings.');
    % This requirement comes from the complex case, where the number
of inputs
    % is 2 times the number of dac timings. I think this could be
tidied up.
    return
end
nis = ni/ndac; % Number of inputs grouped together with a common DAC
timing
           % (2 for the complex case)
if ~nosum   % Sum the responses due to each input set
    y = zeros(tfinal/dt+1,size(S.c,1),nis);
else
    y = zeros(tfinal/dt+1,size(S.c,1),ni);
end
for i = 1:ndac
    n1 = round(tp(i,1)/delta_t);
    n2 = round(tp(i,2)/delta_t);
    z1 = [n1 size(y1,2) nis];
    z2 = [n2 size(y1,2) nis];
    yy = [zeros(z1); y1(1:nf-n1+1,:,(i-1)*nis+1:i*nis)] ...
       - [zeros(z2); y1(1:nf-n2+1,:,(i-1)*nis+1:i*nis)];
    yy = yy(1:nd:end,:,:);
    if ~nosum   % Sum the responses due to each input set
    y = y + yy;
```

```
    else
    y(:,:,i) = yy;
    end
end
```

realizeNTF_ct.m

```
function [ABCDc,tdac2] = realizeNTF_ct( ntf, form, tdac, ordering, bp,
ABCDc)
% [ABCDc,tdac2] = realizeNTF_ct( ntf, form='FB', tdac, ordering=[1:n],
bp=zeros(...), ABCDc)
% Realize an NTF with a continuous-time loop filter.
%
% Output
% ABCDc      A state-space description of the CT loop filter
%
% tdac2      A matrix with the DAC timings, including ones
%        that were automatically added.
%
%
% Input Arguments
% ntf   A noise transfer function in pole-zero form.
%
% form = {'FB','FF'}
%       A string specifying the topology of the loop filter.
%   For the FB structure, the elements of Bc are calculated
%   so that the sampled pulse response matches the L1 impulse
%   respnse.  For the FF structure, Cc is calculated.
%
% tdac  The timing for the feedback DAC(s). If tdac(1)>=1,
%   direct feedback terms are added to the quantizer.
%   Multiple timings (1 or more per integrator) for the FB
%       topology can be specified by making tdac a cell array,
%       e.g. tdac = { [1,2]; [1 2]; {[0.5 1],[1 1.5]}; []};
%   In this example, the first two integrators have
%   dacs with [1,2] timing, the third has a pair of
%   dacs, one with [0.5 1] timing and the other with
%   [1 1.5] timing, and there is no direct feedback
%   DAC to the quantizer
%
% ordering
%   A vector specifying which NTF zero-pair to use in each resonator
%   Default is for the zero-pairs to be used in the order specified in
%       the NTF.
%
% bp    A vector specifying which resonator sections are bandpass.
%   The default (zeros(...)) is for all sections to be lowpass.
%
% ABCDc The loop filter structure, in state-space form.
%   If this argument is omitted, ABCDc is constructed according
```

```
%       to "form."
%

% Handle the input arguments
parameters = {'ntf';'form';'tdac';'ordering';'bp';'ABCDc'};
defaults = {NaN, 'FB', [0 1], [], [], []};
for i=1:length(defaults)
    parameter = char(parameters(i));
    if i>nargin | ( eval(['isnumeric(' parameter ') ']) & ...
     eval(['any(isnan(' parameter ')) | isempty(' parameter ') ']) )
        eval([parameter '=defaults{i};'])
    end
end
ntf_p = ntf.p{1};
ntf_z = ntf.z{1};

order = length(ntf_p);
order2 = floor(order/2);
odd = order - 2*order2;

% compensate for limited accuracy of zero calculation
ntf_z(find(abs(ntf_z - 1) < eps^(1/(1+order)))) = 1;

if iscell(tdac)
    if size(tdac) ~= [order+1 1]
    msg = sprintf(['%s error. For cell array tdac, size(tdac) ' ...
      'must be [order+1 1].\n'],  mfilename);
    error(msg);
    end
    if form ~= 'FB'
    msg = sprintf(['%s error. Currently only supporting form=''FB'' '
...
      'for cell-array tdac'], mfilename);
    error(msg);
    end
else
    if size(tdac) ~= [1 2]
    msg = sprintf(['%s error. For non cell array tdac, size(tdac) '
...
      'must be [1 2].\n'],  mfilename);
    error(msg);
    end
end
if isempty(ordering)
    ordering = [1:order2];
end
if isempty(bp)
    bp = zeros(1,order2);
end
if ~iscell(tdac)
    % Need direct terms for every interval of memory in the DAC
```

```
    n_direct = ceil(tdac(2))-1;
    if ((tdac(1)>0) && (tdac(1)<1) && (tdac(2)>1) && (tdac(2)<2))
    n_extra = n_direct-1;     % tdac pulse spans a sample point
    else
    n_extra = n_direct;
    end
    tdac2 = [ -1 -1;
           tdac;
           0.5*ones(n_extra,1)*[-1 1] + cumsum(ones(n_extra,2),1) ...
                                        + (n_direct - n_extra) ];
else
    n_direct = 0;
    n_extra = 0;
end

if isempty(ABCDc)
    ABCDc = zeros(order+1,order+2);
    % Stuff the A portion
    if odd
    ABCDc(1,1) = real( log( ntf_z(1) ) );
    ABCDc(2,1) = 1;
    end
    for i = 1:order2
    n = bp(i);
    i1 = 2*i + odd - 1;
    zi = 2*ordering(i) + odd - 1;
    w = abs( angle( ntf_z(zi) ) );
    ABCDc(i1+[0 1 2],i1+[0 1]) =[ 0  -w^2
                     1   0
                     n  1-n ];
    end
    ABCDc(1,order+1) = 1;
    ABCDc(1,order+2) = -1;  % 2006.10.02 Changed to -1 to make FF STF
have +ve gain at DC
end
Ac = ABCDc(1:order,1:order);
switch form
    case 'FB'
    Cc = ABCDc(order+1,1:order);
    if ~iscell(tdac)
        Bc = [eye(order) zeros(order,1)];
        Dc = [zeros(1,order) 1];
        tp = repmat(tdac,order+1,1);
    else    % Assemble tdac2, Bc and Dc
        tdac2 = [-1 -1];
        Bc = [];
        Dc = [];
        Bci = [eye(order) zeros(order,1)];
        Dci = [zeros(1,order) 1];
        for i=1:length(tdac)
        tdi = tdac{i};
        if iscell(tdi)
```

```
            for j=1:length(tdi)
            tdj = tdi{j};
            tdac2 = [tdac2; tdj];
            Bc = [Bc Bci(:,i)];
            Dc = [Dc Dci(:,i)];
            end
        elseif ~isempty(tdi)
            tdac2 = [tdac2; tdi];
            Bc = [Bc Bci(:,i)];
            Dc = [Dc Dci(:,i)];
        end
        end
        tp = tdac2(2:end,:);
    end
    case 'FF'
    Cc = [eye(order); zeros(1,order)];
    Bc = [-1; zeros(order-1,1)];
    Dc = [zeros(order,1); 1];
    tp = tdac;  % 2008-03-24 fix from Ayman Shabra
    otherwise
    error(sprintf('%s error. Sorry, no code for form "%s".\n', ...
        mfilename, form));
end
% Sample the L1 impulse response
n_imp = ceil( 2*order + max(tdac2(:,2)) +1 );
y = impL1(ntf,n_imp);

sys_c = ss( Ac, Bc, Cc, Dc );
yy = pulse(sys_c,tp,1,n_imp,1);
yy = squeeze(yy);
% Endow yy with n_extra extra impulses.
% These will need to be implemented with n_extra extra DACs.
% !! Note: if t1=int, matlab says pulse(sys) @t1 ~=0
% !! This code corrects this problem.
if n_extra>0
    y_right = padb([zeros(1,n_direct); eye(n_direct)], n_imp+1);
    % Replace the last column in yy with an ordered set of impulses
    if (n_direct > n_extra)
    yy = [yy y_right(:,2:end)];
    else
    yy = [yy(:,1:end-1) y_right];
    end
end

% Solve for the coefficients
x = yy\y;
if norm(yy*x-y) > 1e-4
    warning('Pulse response fit is poor.');
end
switch form
    case 'FB'
    if ~iscell(tdac)
```

```
        Bc2 = [ x(1:order) zeros(order,n_extra) ];
        if (n_extra > 0)
        Dc2 = [ 0 x(order+1:end).'];
        else
        Dc2 = x(order+1:end).';
        end
    else
        BcDc = [Bc;Dc];
        i = find(BcDc);
        BcDc(i) = x;
        Bc2 = BcDc(1:end-1,:);
        Dc2 = BcDc(end,:);
    end
    case 'FF'
    Bc2 = [Bc zeros(order,n_extra)];
    Cc = x(1:order).';
    if (n_extra > 0)
        Dc2 = [ 0 x(order+1:end).'];
    else
        Dc2 = x(order+1:end).';
    end
    otherwise
    fprintf(1,'%s error. No code for form "%s".\n', mfilename, form);
end
Dc1 = 0;
Dc = [Dc1 Dc2];
Bc1 = [1; zeros(order-1,1)];
Bc = [Bc1 Bc2];
% Scale Bc1 for unity STF magnitude at f0
fz = angle(ntf.z{1})/(2*pi);
f1 = fz(1);
ibz = abs(fz-f1) <= abs(fz+f1);
fz = fz(ibz);
f0 = mean(fz);
if min(abs(fz)) < 3*min(abs(fz-f0))
    f0 = 0;
end
L0c = zpk(ss(Ac,Bc1,Cc,Dc1));
G0 = evalTFP(L0c,ntf,f0);
if f0 == 0
    Bc(:,1) = Bc(:,1)*abs(Bc(1,2:end)*(tdac2(2:end,2)-
tdac2(2:end,1))/Bc(1,1));
else
    Bc(:,1) = Bc(:,1)/abs(G0);
end

ABCDc = [Ac Bc; Cc Dc];
ABCDc = ABCDc .* ( abs(ABCDc) > eps^(1/2) ) ;
```

simulateDMS.c

```
/* simulateDSM.c - A MEX-file for simulating a delta-sigma modulator
%[v,xn,xmax,y] = simulateDSM( u, ABCD, nlev[2], x0[0] )
% or
%[v,xn,xmax,y] = simulateDSM( u, ntf, nlev[2], x0[0] )
%
%Compute the output of a general delta-sigma modulator with input u,
%a structure described by ABCD, an initial state x0 (default zero) and
%a quantizer with a number of levels specified by nlev.
%Multiple quantizers are implied by making nlev an array,
% and multiple inputs are implied by the number of rows in u.
%
%Alternatively, the modulator may be described by an NTF.
%The NTF is zpk object. (The STF is assumed to be 1.)
%The structure that is simulated is the block-diagional structure used
by
%zp2ss.m.
%
%The obsolete NTF style is
%a struct with fields 'zeros' and 'poles' ('k' is assumed to be 1).
%the STF is assumed to be 1.
%The structure that is simulated is the block-diagional structure used
by
%zp2ss.m.
*/

#include <stdio.h>
#include <math.h>
#include "mex.h"

/* Global variables */
/* In an effort to make the code more readable and to cut down on the
overhead
   associated with function calls, I have made many variables global.
*/
char *cmdName = "simulateDSM";
int
    order,   /* The order of the modulator. */
    nu,      /* The number of inputs, inferred from size(u,1). */
    nq,      /* The number of quantizers, inferred from nlev */
    N,       /* The number of time steps. */
    ABCD_rows,  /* The number of rows in ABCD */
    saveState;  /* Flag: keep track of the states. */
double
    *u,      /* Points into the input array. */
    *v,      /* Points into the output array. */
    *x,      /* The current state. */
    *xn,     /* Points (in)to the (output) state array. */
    *xMax,   /* Points to the state maxima output array. */
    *py,     /* Points to the quantizer input output array. */
    *ABCD,   /* The ABCD array (col-wise) description of the moduator.
*/
    *nlev,   /* The number of quantizer levels. */
    default_nlev=2;
```

```
#ifdef __STDC__
double quantize(double yy, int nLevels)
#else
double quantize(yy, nLevels)
double yy;
int nLevels;
#endif
{
    double vv;
    if(nLevels%2) { /* Mid-tread quantizer */
    vv = 2*floor(0.5*(yy+1));
    if( vv > nLevels )
        vv = nLevels-1;
    else if( vv < -nLevels )
        vv = 1-nLevels;
    }
    else { /* Mid-rise quantizer */
    vv = 2*floor(0.5*yy)+1;
    if( vv > nLevels )
        vv = nLevels-1;
    else if( vv < -nLevels )
        vv = 1-nLevels;
    }
    return vv;
}

/* The following function is for debugging purposes only */
#ifdef __STDC__
void printMatrix(double *x, int m, int n)
#else
printMatrix(x, m, n)
double *x;
int m, n;
#endif
{
int i,j;
for(i=0; i<m; ++i){
    for(j=0; j<n; ++j)
    mexPrintf("%8.3f ",x[i+m*j]);
    mexPrintf("\n");
    }
}

#ifdef __STDC__
void fatalError(char *s)
#else
fatalError(s)
char *s;
#endif
{
    char msg[1024];
    sprintf(msg, "%s: %s", cmdName, s);
    mexErrMsgTxt(msg);
}

#ifdef __STDC__
void initializeX(const mxArray *M_x0)
#else
initializeX(M_x0)
```

```
mxArray *M_x0;
#endif
{
    int i;
    double *x0 = mxGetPr(M_x0);
    if( mxGetM(M_x0)!=order || mxGetN(M_x0)!=1 )
    fatalError("x0 must be an order x 1 column vector.");
    for(i=0; i<order; ++i)
    x[i] = *x0++;
}

#ifdef __STDC__
void checkArgs(int nlhs, mxArray **plhs, int nrhs, const mxArray
**prhs)
#else
checkArgs(nlhs, plhs, nrhs, prhs)
int nlhs, nrhs;
mxArray *plhs[], *prhs[];
#endif
{
    int i;
    int form;
    double *pABCD;
    const mxArray *arg2=prhs[1];
    mxArray *zeros, *poles;

    /* Verify the rhs (input) arguments */
    if( nrhs < 2 )
    fatalError("At least two input arguments are needed.");
    if( !mxIsDouble(prhs[0]) )
    fatalError("The input vector does not contain double-precision
data.");

    u = mxGetPr(prhs[0]);
    nu = mxGetM(prhs[0]);
    N = mxGetN(prhs[0]);
    nq = 1;
    nlev = &default_nlev;
    if(nrhs>=3)
    if( !( mxIsEmpty(prhs[2]) || mxIsNaN(*mxGetPr(prhs[2])) ) ){
        nq = mxGetM(prhs[2]) * mxGetN(prhs[2]);
        nlev = mxGetPr(prhs[2]);
    }
    /* Determine the form of the modulator */
    if( mxIsClass(arg2,"zpk") ){        /* NTF in zpk form */
    /* Matlab code: [z,p,k] = zpkdata(ntf); zeros = z{1}; poles=p{1}
*/
    mxArray *lhs[3];
    form = 0;
    mexCallMATLAB(3,lhs,1,&arg2,"zpkdata");
    zeros = mxGetCell(lhs[0],0);
    poles = mxGetCell(lhs[1],0);
    if( (order=mxGetNumberOfElements(zeros)) !=
mxGetNumberOfElements(poles) )
        fatalError("The number of poles must equal the number of
zeros.");
    }
    else if( mxIsStruct(arg2) ){        /* Obsolete NTF form */
    if( (zeros=mxGetField(arg2,0,"zeros"))==0 )
```

```
        fatalError("No zeros field in the NTF struct.");
    if( !mxIsNumeric(zeros) )
        fatalError("The zeros field is not numeric.");
    if( (poles=mxGetField(arg2,0,"poles"))==0 )
        fatalError("No poles field in the NTF struct.");
    if( !mxIsNumeric(poles) )
        fatalError("The poles field is not numeric.");
    if( (order=mxGetNumberOfElements(zeros)) !=
mxGetNumberOfElements(poles) )
        fatalError("The number of poles must equal the number of
zeros.");
    mexWarnMsgTxt("You appear to be using an old-style form of NTF
specification.\nAutomatic converstion to the new form will be done for
this release only.");
    form = 2;
    }
    else if( mxIsNumeric(arg2) ){
    if( mxGetN(arg2)==mxGetM(arg2)+nu && mxIsDouble(arg2) ){
        form = 1;           /* ABCD form */
        order = mxGetM(arg2)-nq;
    }
    else if( mxGetN(arg2)==2 ){
        mexWarnMsgTxt("You appear to be using the old-style form of
NTF specification.\nAutomatic converstion to the new form will be done
for this release only.");
        form = 3;       /* old NTF form */
        order = mxGetM(arg2);
    }
    else
        fatalError("ABCD must be an order+nq by order+nu+nq matrix.");
    }
    else
    fatalError("The second argument is neither a proper ABCD matrix
nor an NTF.");

    if( form==1 ){      /* ABCD form */
    ABCD = mxGetPr(arg2);
    }
    else if( form==0 || form==2 || form==3 ){   /* NTF form */
    /* MATLAB code for computing ABCD
       [A,B2,C,D2] = zp2ss(ntf.poles,ntf.zeros,-1);
       D2 = 0;
       % !!!! Assume stf=1
       B1 = -B2;
       D1 = 1;
       ABCD = [ A B1 B2; C D1 D2]
    */
    mxArray *lhs[4],*pntf[3];
    double *p1,*p2;
    if( nu != 1 )
        fatalError("Fatal error. Number of inputs must be 1 for a
modulator specified by its NTF\n");
    if( nq != 1)
        fatalError("Fatal error. Number of quantizers must be 1 for a
modulator specified by its NTF\n");
    /* Copy the ntf (arg2) into the temporary pntf[] matrices. */
    /* This could be accomplished more efficiently (but more
dangerously)
       by directly setting the elements of the mxArray data structure.
*/
```

```
    pntf[0] = mxCreateDoubleMatrix(order,1,mxCOMPLEX);
    pntf[1] = mxCreateDoubleMatrix(order,1,mxCOMPLEX);
    pntf[2] = mxCreateDoubleMatrix(1,1,mxREAL);
    p2 = mxGetPr((form==0||form==2) ? zeros:arg2);
    for( p1=mxGetPr(pntf[1]), i=0; i<order; ++i)
        *p1++ = *p2++;
    if( form!=3 )
        p2 = mxGetPr(poles);
    for( p1=mxGetPr(pntf[0]), i=0; i<order; ++i)
        *p1++ = *p2++;
    p2 = mxGetPi((form==0||form==2) ? zeros:arg2);
    if( p2 ) /* Non-null imaginary part. */
        for( p1=mxGetPi(pntf[1]),  i=0; i<order; ++i)
        *p1++ = *p2++;
    if( form!=3 )
        p2 = mxGetPi(poles);
    if( p2 ) /* Non-null imaginary part. */
        for( p1=mxGetPi(pntf[0]), i=0; i<order; ++i)
        *p1++ = *p2++;
    *mxGetPr(pntf[2]) = -1;

    mexCallMATLAB(4,lhs,3,pntf,"zp2ss");

    p1 = mxGetPr(lhs[0]);
    p2 = mxGetPr(lhs[2]);
    ABCD = (double
*)mxCalloc((order+nu)*(order+nu+nq),sizeof(double));
    pABCD = ABCD;
    for( i=0; i<order; ++i ){
        int j;
        for( j=0; j<order; ++j )
        *pABCD++ = *p1++;
        *pABCD++ = *p2++;
    }
    p1 = mxGetPr(lhs[1]);   /* B1 = -B2 */
    for( i=0; i<order; ++i )
        *pABCD++ = -*p1++;
    *pABCD++ = 1;        /* D1 */
    p1 = mxGetPr(lhs[1]);   /* B2 */
    for( i=0; i<order; ++i )
        *pABCD++ = *p1++;
    *pABCD = 0;      /* D2 */

    for(i=0;i<4;++i)
        mxDestroyArray(lhs[i]);
    for(i=0;i<2;++i)
        mxDestroyArray(pntf[i]);
    }
    else
    fatalError("Internal error. form != 0, 1, 2 or 3!");
    ABCD_rows = order + nq;

    plhs[0] = mxCreateDoubleMatrix(nq,N,mxREAL);
    v = mxGetPr(plhs[0]);

    x = (double *)mxCalloc(order,sizeof(double));
    if(nrhs>=4){
    if( !( mxIsEmpty(prhs[3]) || mxIsNaN(*mxGetPr(prhs[3])) ) )
        initializeX(prhs[3]);
```

```
    }

    /* Verify the lhs (output) arguments */
    saveState=0;
    py=0;
    xMax=0;
    switch(nlhs){
    case 4:
        plhs[3] = mxCreateDoubleMatrix(nq,N,mxREAL);
        py = mxGetPr(plhs[3]);
    case 3:
        plhs[2] = mxCreateDoubleMatrix(order,1,mxREAL);
        xMax = mxGetPr(plhs[2]);
    case 2:
        plhs[1] = mxCreateDoubleMatrix(order,N,mxREAL);
        xn = mxGetPr(plhs[1]);
        saveState=1;
        break;
    case 1:
        break;
    default:
        fatalError("Incorrect number of output arguments.");
    }
    if( !saveState )
    xn = (double *)mxCalloc(order,sizeof(double));
}


/* Simulate the modulator using the difference equations. */
/* For efficiency, store the state in xn and compute from x. */
/* (These variables may be recycled internally, depending
   on the output variables requested.) */

#ifdef __STDC__
void simulateDSM()
#else
simulateDSM()
#endif
{
    int i,j,t, qi;
    double *pABCD, *ptr, *pxn, tmp;

    for( t=0; t<N; ++t ){ /* [xn;y] = ABCD*[x;u;v]; x=xn;*/
    /* Compute y = C*x + D1*u and thence v for each quantizer */
    for( qi=0; qi<nq; ++qi){
        tmp = 0;
        for(i=0, pABCD=ABCD+order+qi, ptr=x; i<order; ++i,
pABCD+=ABCD_rows)
        tmp += (*pABCD) * *ptr++;
        for(i=0, ptr=u; i<nu; ++i, pABCD+=ABCD_rows)
        tmp += (*pABCD) * *ptr++;
        if( py!=0 )
        *py++ = tmp;
        v[qi] = quantize(tmp, nlev[qi]);
    }

    /* Next compute xn = A*x + B*[u;v], */
    for( i=0, pxn=xn; i<order; ++i ){
        tmp=0;
```

```
        pABCD=ABCD+i;
        for( ptr=x, j=0; j<order; ++j, pABCD += ABCD_rows )
        tmp += *pABCD * *ptr++;
        for( ptr=u, j=0; j<nu; ++j, pABCD += ABCD_rows )
        tmp += *pABCD * *ptr++;
        for( ptr=v, j=0; j<nq; ++j, pABCD += ABCD_rows )
        tmp += *pABCD * *ptr++;
        *pxn++ = tmp;
        }
    u += nu;
    v += nq;
    if(xMax!=0){
        for( i=0; i<order; ++i){
        double abs=fabs(xn[i]);
        if( abs > xMax[i] )
            xMax[i] = abs;
        }
    }
    if(saveState){
        x = xn;
        xn += order;
    }
    else { /* swap x and xn */
        double *xtmp = x;
        x = xn;
        xn = xtmp;
    } /* if(saveState) */
    } /* for( t=0 .... ) */
}


#ifdef __STDC__
void mexFunction(int nlhs, mxArray **plhs, int nrhs, const mxArray
**prhs)
#else
mexFunction(nlhs, plhs, nrhs, prhs)
int nlhs, nrhs;
mxArray *plhs[], *prhs[];
#endif
{
    checkArgs(nlhs, plhs, nrhs, prhs);
    /* Print the variables being used
    mexPrintf("x=\n");      printMatrix(x,order,1);
    mexPrintf("\nABCD=\n"); printMatrix(ABCD,order+nq,order+nu+nq);
    */
    simulateDSM();
}
```

simulateDMS.m

```
function [v,xn,xmax,y] = simulateDSM(u,arg2,nlev,x0)
%[v,xn,xmax,y] = simulateDSM(u,ABCD,nlev=2,x0=0)
% or
%[v,xn,xmax,y] = simulateDSM(u,ntf,nlev=2,x0=0)
%
%Compute the output of a general delta-sigma modulator with input u,
%a structure described by ABCD, an initial state x0 (default zero) and
%a quantizer with a number of levels specified by nlev.
%Multiple quantizers are implied by making nlev an array,
%and multiple inputs are implied by the number of rows in u.
%
%Alternatively, the modulator may be described by an NTF.
%The NTF is zpk object. (The STF is assumed to be 1.)
%The structure that is simulated is the block-diagional structure used
by
%zp2ss.m.

fprintf(1,'Warning: You are running the non-mex version of
simulateDSM.\n');
fprintf(1,'Please compile the mex version with "mex
simulateDSM.c"\n');

if nargin<2
    fprintf(1,'Error. simulateDSM needs at least two arguments.\n');
    return
end

% Handle the input arguments
parameters = {'u','arg2','nlev','x0'};
defaults = [ NaN NaN 2 NaN ];
for i=1:length(defaults)
    parameter = char(parameters(i));
    if i>nargin | ( eval(['isnumeric(' parameter ') ']) &  ...
     eval(['any(isnan(' parameter ')) | isempty(' parameter ') ']) )
    eval([parameter '=defaults(i);'])
    end
end
nu = size(u,1);
nq = length(nlev);
if isobject(arg2) & strcmp(class(arg2),'zpk')
    ntf.k = arg2.k;
    ntf.zeros = arg2.z{:};
    ntf.poles = arg2.p{:};
    form = 2;
    order = length(ntf.zeros);
elseif isstruct(arg2)
    if any(strcmp(fieldnames(arg2),'zeros'))
    ntf.zeros = arg2.zeros;
    else
    error('No zeros field in the NTF.')
    end
    if any(strcmp(fieldnames(arg2),'poles'))
    ntf.poles = arg2.poles;
    else
```

```
    error('No poles field in the NTF.')
    end
    form = 2;
    order = length(ntf.zeros);
elseif isnumeric(arg2)
    if size(arg2,2) > 2 & size(arg2,2)==nu+size(arg2,1) % ABCD
dimesions OK
    form = 1;
    ABCD = arg2;
    order = size(ABCD,1)-nq;
    else
    fprintf(1,'The ABCD argument does not have proper dimensions.\n');
    if size(arg2,2) == 2        % Probably old (ver. 2) ntf form
        fprintf(1,'You appear to be using the old-style form of NTF
specification.\n Automatic converstion to the new form will be done
for this release only.\n');
        ntf.zeros = arg2(:,1);
        ntf.poles = arg2(:,2);
        form = 2;
        order = length(ntf.zeros);
    else
        error('Exiting simulateDSM.')
    end
    end
else
    error('The second argument is neither an ABCD matrix nor an
NTF.\n');
end
if isnan(x0)
    x0 = zeros(order,1);
end

if form==1
    A = ABCD(1:order, 1:order);
    B = ABCD(1:order, order+1:order+nu+nq);
    C = ABCD(order+1:order+nq, 1:order);
    D1= ABCD(order+1:order+nq, order+1:order+nu);
else
    [A,B2,C,D2] = zp2ss(ntf.poles,ntf.zeros,-1);    % A realization of
1/H
    % Transform the realization so that C = [1 0 0 ...]
    Sinv = orth([C' eye(order)])/norm(C); S = inv(Sinv);
    C = C*Sinv;
    if C(1)<0
    S = -S;
    Sinv = -Sinv;
    end
    A = S*A*Sinv; B2 = S*B2; C = [1 zeros(1,order-1)]; % C=C*Sinv;
    D2 = 0;
    % !!!! Assume stf=1
    B1 = -B2;
    D1 = 1;
    B = [B1 B2];
end

N = length(u);
v = zeros(nq,N);
y = zeros(nq,N);
if nargout > 1  % Need to store the state information
    xn = zeros(order,N);
```

```
end
if nargout > 2  % Need to keep track of the state maxima
    xmax = abs(x0);
end

for i=1:N
    y(:,i) = C*x0 + D1*u(:,i);
    v(:,i) = ds_quantize(y(:,i),nlev);
    x0 = A * x0 + B * [u(:,i);v(:,i)];
    if nargout > 1  % Save the next state
    xn(:,i) = x0;
    end
    if nargout > 2  % Keep track of the state maxima
    xmax = max(abs(x0),xmax);
    end
end
return

function v = ds_quantize(y,n)
%v = ds_quantize(y,n)
%Quantize y to
% an odd integer in [-n+1, n-1], if n is even, or
% an even integer in [-n, n], if n is odd.
%
%This definition gives the same step height for both mid-rise
%and mid-tread quantizers.

if rem(n,2)==0  % mid-rise quantizer
    v = 2*floor(0.5*y)+1;
else        % mid-tread quantizer
    v = 2*floor(0.5*(y+1));
end

% Limit the output
for qi=1:length(n)  % Loop for multiple quantizers
    L = n(qi)-1;
    i = v(qi,:)>L;
    if any(i)
    v(qi,i) = L;
    end
    i = v(qi,:)<-L;
    if any(i)
    v(qi,i) = -L;
    end
end
```

simulateSNR.m

```
function [snr,amp] = simulateSNR(arg1,osr,amp,f0,nlev,f,k,quadrature)
%[snr,amp] =
simulateSNR(ntf|ABCD|function,osr,amp,f0=0,nlev=2,f=1/(4*osr),k=13,qua
drature=0)
%Determine the SNR for a delta-sigma modulator by using simulations.
%The modulator is described by a noise transfer function (ntf)
%and the number of quantizer levels (nlev).
%Alternatively, the first argument to simulateSNR may be an ABCD
matrix or
%the name of a function taking the input signal as its sole argument.
%The band of interest is defined by the oversampling ratio (osr)
%and the center frequency (f0).
%The input signal is characterized by the amp vector and the f
variable.
%amp defaults to [-120 -110...-20 -15 -10 -9 -8 ... 0]dB, where 0 dB
means
%a full-scale (peak value = nlev-1) sine wave.
%f is the input frequency, normalized such that 1 -> fs;
%f is rounded to an FFT bin.
%
%Using sine waves located in FFT bins, the SNR is calculated as the
ratio
%of the sine wave power to the power in all in-band bins other than
those
%associated with the input tone. Due to spectral smearing, the input
tone
%is not allowed to lie in bins 0 or 1. The length of the FFT is 2^k.
%
% If ntf is complex, simulateQDSM (which is slow) is called.
% If ABCD is complex, simulateDSM is used with the real equivalent of
ABCD
% in order to speed up simulations.

% Future versions may accommodate STFs.

% Handle the input arguments
if nargin<1
    error('Insufficient arguments');
end
parameters = {'arg1';'osr';'amp';'f0';'nlev';'f';'k';'quadrature'};
defaults = {NaN 64 NaN 0 2 NaN 13 0};
for arg_i=1:length(defaults)
    parameter = char(parameters(arg_i));
    if arg_i>nargin | ( eval(['isnumeric(' parameter ') ']) &  ...
     eval(['any(isnan(' parameter ')) | isempty(' parameter ') ']) )
        eval([parameter '=defaults{arg_i};'])
    end
end
% Look at arg1 and decide if the system is quadrature
quadrature_ntf = 0;
if ischar(arg1)
    is_function = 1;
else
    is_function = 0;
```

```
    if isobject(arg1) % zpk object
        pz = [arg1.p{1} arg1.z{1}];
        for i=1:2
            if any( abs(imag(poly(pz(:,i)))) > 1e-4 )
                quadrature = 1;
                quadrature_ntf = 1;
            end
        end
    else            % ABCD matrix
        if ~all(all(imag(arg1)==0))
            quadrature = 1;
        end
    end
end
if isnan(amp)
    amp = [-120:10:-20 -15 -10:0];
end
osr_mult = 2;
if f0~=0 & ~quadrature
    osr_mult = 2*osr_mult;
end
if isnan(f)
    f = f0 + 0.5/(osr*osr_mult);    % Halfway across the band
end
M = nlev-1;
if quadrature & ~quadrature_ntf
    % Modify arg1 (ABCD) and nlev so that simulateDSM can be used
    nlev = [nlev; nlev];
    arg1 = mapQtoR(arg1);
end

if abs(f-f0) > 1/(osr*osr_mult)
    fprintf(1,'Warning: the input tone is out-of-band.\n');
end

N = 2^k;
if N < 8*2*osr  % Require at least 8 bins to be "in-band"
    fprintf(1,'Warning: Increasing k to accommodate a large
oversampling ratio.\n');
    k = ceil(log2(8*2*osr))
    N = 2^k;
end
F = round(f*N);
if abs(F)<=1
    fprintf(1,'Warning: Increasing k to accommodate a low input
frequency.\n');
    % Want f*N >= 1
    k = ceil(log2(1/f))
    N = 2^k;
    F = 2;
end

Ntransient = 100;
soft_start = 0.5*(1-cos(2*pi/Ntransient*[0:Ntransient/2-1]));
if ~quadrature
    tone = M * sin(2*pi*F/N*[0:(N+Ntransient-1)]);
    tone(1:Ntransient/2) = tone(1:Ntransient/2) .* soft_start;
else
    tone = M * exp(2i*pi*F/N*[0:(N+Ntransient-1)]);
```

```
    tone(1:Ntransient/2) = tone(1:Ntransient/2) .* soft_start;
    if ~quadrature_ntf
        tone = [real(tone); imag(tone)];
    end
end
window = .5*(1 - cos(2*pi*(0:N-1)/N) );      %Hann window
if f0==0
    % Exclude DC and its adjacent bin
    inBandBins = N/2+[3:round(N/(osr_mult*osr))];
    F = F-2;
else
    f1 = round(N*(f0-1/(osr_mult*osr)));
    inBandBins = N/2+[f1:round(N*(f0+1/(osr_mult*osr)))]; % Should
exclude DC
    F = F-f1+1;
end

snr = zeros(size(amp));
i=1;
for A = 10.^(amp/20);
    if is_function
        v = feval(arg1, A*tone);
    elseif quadrature_ntf
        v = simulateQDSM(A*tone, arg1, nlev);
    else
        v = simulateDSM(A*tone, arg1, nlev);
        if quadrature
            v = v(1,:) + 1i*v(2,:);
        end
    end
    hwfft = fftshift(fft(window.*v(1+Ntransient:N+Ntransient)));
    snr(i) = calculateSNR(hwfft(inBandBins),F);
    i=i+1;
end
```

synthesizeNTF.m

```
function ntf = synthesizeNTF(order,osr,opt,H_inf,f0)
%ntf = synthesizeNTF(order=3,osr=64,opt=0,H_inf=1.5,f0=0)
%Synthesize a noise transfer function for a delta-sigma modulator.
%   order = order of the modulator
%   osr =   oversampling ratio
%   opt =   flag for optimized zeros
%       0 -> not optimized,
%       1 -> optimized,
%       2 -> optimized with at least one zero at band-center
%       3 -> optimized zeros (Requires MATLAB6 and Optimization
Toolbox)
%       [z] -> zero locations in complex form
%   H_inf = maximum NTF gain
%   f0 =    center frequency (1->fs)
%
%ntf is a zpk object containing the zeros and poles of the NTF. See
zpk.m
% See also
```

```
%  clans()    "Closed-loop analysis of noise-shaper." An alternative
%             method for selecting NTFs based on the 1-norm of the
%             impulse response of the NTF
%
%  synthesizeChebyshevNTF()    Select a type-2 highpass Chebyshev NTF.
%             This function does a better job than synthesizeNTF if osr
%             or H_inf is low.

% This is actually a wrapper function which calls either the
% appropriate version of synthesizeNTF based on the availability
% of the 'fmincon' function from the Optimization Toolbox
% Handle the input arguments
parameters = {'order' 'osr' 'opt' 'H_inf' 'f0'};
defaults = { 3 64 0 1.5 0 };
for arg_i=1:length(defaults)
    parameter = char(parameters(arg_i));
    if arg_i>nargin | ( eval(['isnumeric(' parameter ') ']) &  ...
     eval(['any(isnan(' parameter ')) | isempty(' parameter ') ']) )
        eval([parameter '=defaults{arg_i};'])
    end
end
if f0 > 0.5
    fprintf(1,'Error. f0 must be less than 0.5.\n');
    return;
end
if f0 ~= 0 & f0 < 0.25/osr
    warning('(%s) Creating a lowpass ntf.', mfilename);
    f0 = 0;
end
if f0 ~= 0 & rem(order,2) ~= 0
    fprintf(1,'Error. order must be even for a bandpass
modulator.\n');
    return;
end

if length(opt)>1 & length(opt)~=order
    fprintf(1,'The opt vector must be of length %d(=order).\n',
order);
    return;
end



if exist('fmincon','file')
    ntf = synthesizeNTF1(order,osr,opt,H_inf,f0);
else
    ntf = synthesizeNTF0(order,osr,opt,H_inf,f0);
end
```

synthesizeNTF0.m

```
function ntf = synthesizeNTF0(order,OSR,opt,H_inf,f0)

% Determine the zeros.
if f0~=0        % Bandpass design-- halve the order temporarily.
    order = order/2;
    dw = pi/(2*OSR);
else
    dw = pi/OSR;
end

if length(opt)==1
    if opt==0
    z = zeros(order,1);
    else
    z = dw*ds_optzeros(order,opt);
    if isempty(z)
            return;
    end
    end
    if f0~=0        % Bandpass design-- shift and replicate the zeros.
    order = order*2;
    z = z + 2*pi*f0;
    ztmp = [ z'; -z' ];
    z = ztmp(:);
    end
    z = exp(j*z);
else
    z = opt(:);
end

ntf = zpk(z,zeros(1,order),1,1);
itn_limit = 100;

% Iteratively determine the poles by finding the value of the x-
parameter
% which results in the desired H_inf.
if f0 == 0          % Lowpass design
    HinfLimit = 2^order;  % !!! The limit is actually lower for opt=1
and low OSR
    if H_inf >= HinfLimit
    fprintf(2,'%s warning: Unable to achieve specified Hinf.\n',
mfilename);
    fprintf(2,'Setting all NTF poles to zero.\n');
    ntf.p = zeros(order,1);
    else
    x=0.3^(order-1);    % starting guess
    converged = 0;
```

```
    for itn=1:itn_limit
        me2 = -0.5*(x^(2./order));
        w = (2*[1:order]'-1)*pi/order;
        mb2 = 1+me2*exp(j*w);
        p = mb2 - sqrt(mb2.^2-1);
        out = find(abs(p)>1);
        p(out) = 1./p(out); % reflect poles to be inside the unit
circle.
        ntf.z = z;  ntf.p = cplxpair(p);
        f = real(evalTF(ntf,-1))-H_inf;
    % [ x f ]
        if itn==1
        delta_x = -f/100;
        else
        delta_x = -f*delta_x/(f-fprev);
        end

        xplus = x+delta_x;
        if xplus>0
        x = xplus;
        else
        x = x*0.1;
        end
        fprev = f;

        if abs(f)<1e-10 | abs(delta_x)<1e-10
        converged = 1;
        break;
        end
        if x>1e6
        fprintf(2,'%s warning: Unable to achieve specified Hinf.\n',
mfilename);
        fprintf(2,'Setting all NTF poles to zero.\n');
        ntf.z = z;  ntf.p = zeros(order,1);
        break;
        end
        if itn == itn_limit
        fprintf(2,'%s warning: Danger! Iteration limit
exceeded.\n',...
          mfilename);
        end
    end
    end
else                % Bandpass design.
    x = 0.3^(order/2-1);    % starting guess (not very good for f0~0)
    if f0>0.25
    z_inf=1;
    else
    z_inf=-1;
    end
    c2pif0 = cos(2*pi*f0);
    for itn=1:itn_limit
```

```
    e2 = 0.5*x^(2./order);
    w = (2*[1:order]'-1)*pi/order;
    mb2 = c2pif0 + e2*exp(j*w);
    p = mb2 - sqrt(mb2.^2-1);
    % reflect poles to be inside the unit circle.
    out = find(abs(p)>1);
    p(out) = 1./p(out);
    ntf.z = z;  ntf.p = cplxpair(p);
    f = real(evalTF(ntf,z_inf))-H_inf;
%   [x f]
    if itn==1
        delta_x = -f/100;
    else
        delta_x = -f*delta_x/(f-fprev);
    end

    xplus = x+delta_x;
    if xplus > 0
        x = xplus;
    else
        x = x*0.1;
    end

    fprev = f;
    if abs(f)<1e-10 | abs(delta_x)<1e-10
        break;
    end
    if x>1e6
        fprintf(2,'%s warning: Unable to achieve specified Hinf.\n',
mfilename);
        fprintf(2,'Setting all NTF poles to zero.\n');
        ntf.p = zeros(order,1);
        break;
    end
    if itn == itn_limit
        fprintf(2,'%s warning: Danger! Iteration limit
exceeded.\n',...
          mfilename);
    end
    end
end

z = cplxpair(ntf.z{:});
p = cplxpair(ntf.p{:});
rev = order:-1:1;

% Assemble the ntf struct
ntf.z = z(rev)';
ntf.p = p(rev)';
```

synthesizeNTF1.m

```
function ntf = synthesizeNTF1(order,osr,opt,H_inf,f0)

% Determine the zeros.
if f0~=0        % Bandpass design-- halve the order temporarily.
    order = order/2;
    dw = pi/(2*osr);
else
    dw = pi/osr;
end

if length(opt)==1
    if opt==0
        z = zeros(order,1);
    else
        z = dw*ds_optzeros(order,1+rem(opt-1,2));
        if isempty(z)
            return;
        end
    end
    if f0~=0        % Bandpass design-- shift and replicate the zeros.
        order = order*2;
        z = sort(z) + 2*pi*f0;
        ztmp = [ z'; -z' ];
        z = ztmp(:);
    end
    z = exp(1i*z);
else
    z = opt(:);
end

zp = z(angle(z)>0);
x0 = (angle(zp)-2*pi*f0) * osr / pi;
if opt==4 && f0~=0
    % Do not optimize the zeros at f0
    x0( abs(x0)<1e-10 ) = [];
end

ntf = zpk(z,zeros(1,order),1,1);
Hinf_itn_limit = 100;

opt_iteration = 5;  % Max number of zero-optimizing/Hinf iterations
while opt_iteration > 0
    % Iteratively determine the poles by finding the value of the x-
parameter
    % which results in the desired H_inf.
    ftol = 1e-10;
```

```
    if f0>0.25
        z_inf=1;
    else
        z_inf=-1;
    end
    if f0 == 0          % Lowpass design
        HinfLimit = 2^order;  % !!! The limit is actually lower for
opt=1 and low osr
        if H_inf >= HinfLimit
            fprintf(2,'%s warning: Unable to achieve specified
Hinf.\n', mfilename);
            fprintf(2,'Setting all NTF poles to zero.\n');
            ntf.p = zeros(order,1);
        else
            x=0.3^(order-1);    % starting guess
            for itn=1:Hinf_itn_limit
                me2 = -0.5*(x^(2./order));
                w = (2*(1:order)'-1)*pi/order;
                mb2 = 1+me2*exp(1i*w);
                p = mb2 - sqrt(mb2.^2-1);
                out = find(abs(p)>1);
                p(out) = 1./p(out); % reflect poles to be inside the
unit circle.
                p = cplxpair(p);
                ntf.z = z;  ntf.p = p;
                f = real(evalTF(ntf,z_inf))-H_inf;
                % [ x f ]
                if itn==1
                    delta_x = -f/100;
                else
                    delta_x = -f*delta_x/(f-fprev);
                end

                xplus = x+delta_x;
                if xplus>0
                    x = xplus;
                else
                    x = x*0.1;
                end
                fprev = f;
                if abs(f)<ftol || abs(delta_x)<1e-10
                    break;
                end
                if x>1e6
                    fprintf(2,'%s warning: Unable to achieve specified
Hinf.\n', mfilename);
                    fprintf(2,'Setting all NTF poles to zero.\n');
                    ntf.z = z;  ntf.p = zeros(order,1);
                    break;
                end
                if itn == Hinf_itn_limit
                    fprintf(2,'%s warning: Danger! Iteration limit
```

```
exceeded.\n',...
                        mfilename);
                end
            end
        end
    else                % Bandpass design.
        x = 0.3^(order/2-1);    % starting guess (not very good for
f0~0)
        c2pif0 = cos(2*pi*f0);
        for itn=1:Hinf_itn_limit
            e2 = 0.5*x^(2./order);
            w = (2*(1:order)'-1)*pi/order;
            mb2 = c2pif0 + e2*exp(1i*w);
            p = mb2 - sqrt(mb2.^2-1);
            % reflect poles to be inside the unit circle.
            out = find(abs(p)>1);
            p(out) = 1./p(out);
            p = cplxpair(p);
            ntf.z = z;  ntf.p = p;
            f = real(evalTF(ntf,z_inf))-H_inf;
            %   [x f]
            if itn==1
                delta_x = -f/100;
            else
                delta_x = -f*delta_x/(f-fprev);
            end

            xplus = x+delta_x;
            if xplus > 0
                x = xplus;
            else
                x = x*0.1;
            end
            fprev = f;
            if abs(f)<ftol || abs(delta_x)<1e-10
                break;
            end
            if x>1e6
                fprintf(2,'%s warning: Unable to achieve specified
Hinf.\n', mfilename);
                fprintf(2,'Setting all NTF poles to zero.\n');
                p = zeros(order,1);
                ntf.p = p;
                break;
            end
            if itn == Hinf_itn_limit
                fprintf(2,'%s warning: Danger! Hinf iteration limit
exceeded.\n',...
                    mfilename);
            end
        end
    end
```

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    if opt < 3   % Do not optimize the zeros
        opt_iteration = 0;
    else
        if f0 == 0
            ub = ones(size(x0));
            lb = zeros(size(x0));
        else
            ub = 0.5*ones(size(x0));
            lb = -ub;
        end
        options = optimset('TolX',0.001, 'TolFun',0.01, 'MaxIter',100
);
        options = optimset(options,'LargeScale','off');
        options = optimset(options,'Algorithm','active-set');
        options = optimset(options,'Display','off');
        %options = optimset(options,'Display','iter');
        vn = sscanf(version,'%d');
        if vn>=6
            x = fmincon( @(x) ds_synNTFobj1(x,p,osr,f0),
x0,[],[],[],[], lb,ub,[],options);
        else
            error('To use opt>=3 you need to have the Optimization
Toolbox and Matlab 6 or higher');
        end
        x0 = x;
        z = exp(2i*pi*(f0+0.5/osr*x));
        if f0>0
            z = padt(z,length(p)/2,exp(2i*pi*f0));
        end
        z = [z conj(z)].'; z = z(:);
        if f0==0
            z = padt(z,length(p),1);
        end
        ntf.z = z;  ntf.p = p;
        if  abs( real(evalTF(ntf,z_inf)) - H_inf ) < ftol
            opt_iteration = 0;
        else
            opt_iteration = opt_iteration - 1;
        end
    end
end
```

undbm.m

```
function v=undbm(p,z)
% v=undbm(p,z=50) = sqrt(z*10^(p/10-3)) rms voltage equivalent to a
power p indBm
if nargin<2
    z = 50;
end
v = sqrt(z*10.^(p/10-3));
```

undbp.m

```
function y = undbp(x)
% y = undbp(x)  Convert x from dB to a power
y = 10.^(x/10);
```

undbv.m

```
function y = undbv(x)
% y = undbv(x)  Convert x from dB to a voltage
y = 10.^(x/20);
```

zinc.m

```
function mag = zinc(f,m,n)
% mag = zinc(f,m=64,n=1)    Calculate the magnitude response
% of a cascade of n mth-order comb filters at frequencies f.

if nargin<3
    n = 1;
    if nargin<2
    m = 64;
    end
end

mag = abs( sinc(m*f) ./ sinc(f) ).^n;
```

8 BIBLIOGRAFÍA.

1) Brückner, "Simulation environment for CT Sigma-Delta Modulators using FPGA", IEEE Transactions on Circuits and Systems –II: Express Briefs, vol. 59, no. 8, August 2012.
2) H. Inose and Yasuda, "A unitiy bit codinf method by negative feedback", Proceedings of the IEEE, vol. 51, no. 11, pp. 1524-1535, Nov. 1963.
3) J.C. Candy. "A use of double integration in sigma-delta modulation".*IEEE Trans. Communications*, pp. 249-258, Mar 1985
4) Brückner, T.; Kiebler, M.; Lorenz, M.; Zorn, C.; Anders, J.; Mathis, W.; Ortmanns, M., "*Discrete-Time Simulation of Continuous-Time ΣΔ Modulators With Arbitrary Input Signals",* 20th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Abu Dhabi, UAE, December 2013.